ZMQ 指南

wizardforcel

Published with GitBook



目錄

ZMQ 指南

作者: **Pieter Hintjens ph@imatix.com, CEO iMatix Corporation.**

原文地址: **https://github.com/imatix/zguide/tree/v2.2**

翻译: 张吉 **jizhang@anjuke.com**, 安居客集团 好租网工程师

NOTE: 此翻译涵盖2011年10月份的ZMQ稳定版本，即2.1.0 stable release。但读者仍然可以通过此文了解ZMQ的一些基本概念和哲学。

ZMQ 指南

作者: **Pieter Hintjens ph@imatix.com, CEO iMatix Corporation.**
翻译: 张吉 **jizhang@anjuke.com,** 安居客集团 好租网工程师

With thanks to Bill Desmarais, Brian Dorsey, CAF, Daniel Lin, Eric Desgranges, Gonzalo Diethelm, Guido Goldstein, Hunter Ford, Kamil Shakirov, Martin Sustrik, Mike Castleman, Naveen Chawla, Nicola Peduzzi, Oliver Smith, Olivier Chamoux, Peter Alexander, Pierre Rouleau, Randy Dryburgh, John Unwin, Alex Thomas, Mihail Minkov, Jeremy Avnet, Michael Compton, Kamil Kisiel, Mark Kharitonov, Guillaume Aubert, Ian Barber, Mike Sheridan, Faruk Akgul, Oleg Sidorov, Lev Givon, Allister MacLeod, Alexander D'Archangel, Andreas Hoelzlwimmer, Han Holl, Robert G. Jakabosky, Felipe Cruz, Marcus McCurdy, Mikhail Kulemin, Dr. Gergő Érdi, Pavel Zhukov, Alexander Else, Giovanni Ruggiero, Rick "Technoweenie", Daniel Lundin, Dave Hoover, Simon Jefford, Benjamin Peterson, Justin Case, Devon Weller, Richard Smith, Alexander Morland, Wadim Grasza, Michael Jakl, and Zed Shaw for their contributions, and to Stathis Sideris for Ditaa.

Please use the issue tracker for all comments and errata. This version covers the latest stable release of 0MQ and was published on Mon 10 October, 2011.

The Guide is mainly in C, but also in PHP and Lua.

第一章 ZeroMQ基础

拯救世界

如何解释ZMQ？有些人会先说一堆ZMQ的好：它是一套用于快速构建的套接字组件；它的信箱系统有超强的路由能力；它太快了！而有些人则喜欢分享他们被ZMQ点悟的时刻，那些被灵感击中的瞬间：所有的事情突然变得简单明了，让人大开眼界。另一些人则会拿ZMQ同其他产品做个比较：它更小，更简单，但却让人觉得如此熟悉。对于我个人而言，我则更倾向于和别人分享ZMQ的诞生史，相信会和各位读者有所共鸣。

编程是一门科学，但往往会乔装成一门艺术。我们从不去了解软件最底层的机理，或者说根本没有人在乎这些。软件并不只是算法、数据结构、编程语言、或者抽象云云，这些不过是一些工具而已，被我们创造、使用、最后抛弃。软件真正的本质，其实是人的本质。

举例来说，当我们遇到一个高度复杂的问题时，我们会群策群力，分工合作，将问题拆分为若干个部分，一起解决。这里就体现了编程的科学：创建一组小型的构建模块，让人们易于理解和使用，那么大家就会一起用它来解决问题。

我们生活在一个普遍联系的世界里，需要现代的编程软件为我们做指引。所以，未来我们所需要的用于处理大规模计算的构建模块，必须是普遍联系的，而且能够并行运作。那时，程序代码不能再只关注自己，它们需要互相交流，变得足够健谈。程序代码需要像人脑一样，数以兆计的神经元高速地传输信号，在一个没有中央控制的环境下，没有单点故障的环境下，解决问题。这一点其实并不意外，因为就当今的网络来讲，每个节点其实就像是连接了一个人脑一样。

如果你曾和线程、协议、或网络打过交道，你会觉得我上面的话像是天方夜谭。因为在实际应用过程中，只是连接几个程序或网络就已经非常困难和麻烦了。数以兆计的节点？那真是无法想象的。现今只有资金雄厚的企业才能负担得起这种软件和服务。

当今世界的网络结构已经远远超越了我们自身的驾驭能力。十九世纪八十年代的软件危机，弗莱德•布鲁克斯曾说过，这个世上没有银弹。后来，免费和开源解决了这次软件危机，让我们能够高效地分享知识。如今，我们又面临一次新的软件危机，只不过我们谈论得不多。只有那些大型的、富足的企业才有财力建立高度联系的应用程序。那里有云的存在，但它是私有的。我们的数据和知识正在从我们的个人电脑中消失，流入云端，无法获得或与其竞争。是谁坐拥我们的社交网络？这真像一次巨型主机的革命。

我们暂且不谈其中的政治因素，光那些就可以另外出本书了。目前的现状是，虽然互联网能够让千万个程序相连，但我们之中的大多数却无法做到这些。这样一来，那些真正有趣的大型问题（如健康、教育、经济、交通等领域），仍然无法解决。我们没有能力将代码连接起来，也就不能像大脑中的神经元一样处理那些大规模的问题。

已经有人尝试用各种方法来连接应用程序，如数以千计的IETF规范，每种规范解决一个特定问题。对于开发人员来说，HTTP协议是比较简单和易用的，但这也往往让问题变得更糟，因为它鼓励人们形成一种重服务端、轻客户端的思想。

所以迄今为止人们还在使用原始的TCP/UDP协议、私有协议、HTTP协议、网络套接字等形式连接应用程序。这种做法依旧让人痛苦，速度慢又不易扩展，需要集中化管理。而分布式的P2P协议又仅仅适用于娱乐，而非真正的应用。有谁会使用Skype或者Bittorrent来交换数据呢？

这就让我们回归到编程科学的问题上来。想要拯救这个世界，我们需要做两件事情：一，如何在任何地点连接任何两个应用程序；二、将这个解决方案用最为简单的方式包装起来，供程序员使用。

也许这听起来太简单了，但事实确实如此。

ZMQ简介

ZMQ（ØMQ、ZeroMQ, 0MQ）看起来像是一套嵌入式的网络链接库，但工作起来更像是一个并发式的框架。它提供的套接字可以在多种协议中传输消息，如线程间、进程间、TCP、广播等。你可以使用套接字构建多对多的连接模式，如扇出、

发布-订阅、任务分发、请求-应答等。ZMQ的快速足以胜任集群应用产品。它的异步I/O机制让你能够构建多核应用程序，完成异步消息处理任务。ZMQ有着多语言支持，并能在几乎所有的操作系统上运行。ZMQ是iMatix公司的产品，以LGPL开源协议发布。

需要具备的知识

- 使用最新的ZMQ稳定版本；
- 使用Linux系统或其他相似的操作系统；
- 能够阅读C语言代码，这是本指南示例程序的默认语言；
- 当我们书写诸如PUSH或SUBSCRIBE等常量时，你能够找到相应语言的实现，如ZMQ_PUSH、ZMQ_SUBSCRIBE。

获取示例

本指南的所有示例都存放于github仓库中，最简单的获取方式是运行以下代码：

```
git clone git://github.com/imatix/zguide.git
```

浏览examples目录，你可以看到多种语言的实现。如果其中缺少了某种你正在使用的语言，我们很希望你可以提交一份补充。这也是本指南实用的原因，要感谢所有做出过贡献的人。

所有的示例代码都以MIT/X11协议发布，若在源代码中有其他限定的除外。

提问-回答

让我们从简单的代码开始，一段传统的Hello World程序。我们会创建一个客户端和一个服务端，客户端发送Hello给服务端，服务端返回World。下文是C语言编写的服务端，它在5555端口打开一个ZMQ套接字，等待请求，收到后应答World。

hwserver.c: Hello World server

```
//
//  Hello World 服务端
//  绑定一个REP套接字至tcp://*:5555
//  从客户端接收Hello，并应答World
//
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main (void)
{
    void *context = zmq_init (1);

    //  与客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_bind (responder, "tcp://*:5555");

    while (1) {
        //  等待客户端请求
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_recv (responder, &request, 0);
        printf ("收到 Hello\n");
        zmq_msg_close (&request);

        //  做些“处理”
        sleep (1);

        //  返回应答
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 5);
        memcpy (zmq_msg_data (&reply), "World", 5);
        zmq_send (responder, &reply, 0);
        zmq_msg_close (&reply);
    }
    //  程序不会运行到这里，以下只是演示我们应该如何结束
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

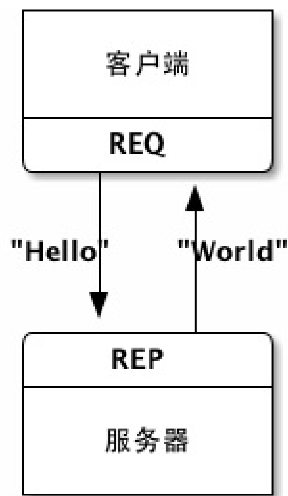


Figure 1 — Request-Reply

使用REQ-REP套接字发送和接受消息是需要遵循一定规律的。客户端首先使用zmq_send()发送消息，再用zmq_recv()接收，如此循环。如果打乱了这个顺序（如连续发送两次）则会报错。类似地，服务端必须先进行接收，后进行发送。

ZMQ使用C语言作为它参考手册的语言，本指南也以它作为示例程序的语言。如果你正在阅读本指南的在线版本，你可以看到示例代码的下方有其他语言的实现。如以下是C++语言：

hwserver.cpp: Hello World server

```
//
// Hello World 服务端 C++语言版
// 绑定一个REP套接字至tcp://*:5555
// 从客户端接收Hello，并应答World
//
#include <zmq.hpp>
#include <string>
#include <iostream>
#include <unistd.h>

int main () {
    // 准备上下文和套接字
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;

        // 等待客户端请求
        socket.recv (&request);
        std::cout << "收到 Hello" << std::endl;

        // 做一些“处理”
        sleep (1);

        // 应答World
        zmq::message_t reply (5);
        memcpy ((void *) reply.data (), "World", 5);
        socket.send (reply);
    }
    return 0;
}
```

可以看到C语言和C++语言的API代码差不多，而在PHP这样的语言中，代码就会更为简洁：

hwserver.php: Hello World server

```php
<?php
/**
 * Hello World 服务端
 * 绑定REP套接字至 tcp://*:5555
 * 从客户端接收Hello，并应答World
 * @author Ian Barber <ian(dot)barber(at)gmail(dot)com>
 */

$context = new ZMQContext(1);

// 与客户端通信的套接字
$responder = new ZMQSocket($context, ZMQ::SOCKET_REP);
$responder->bind("tcp://*:5555");

while(true) {
    // 等待客户端请求
    $request = $responder->recv();
    printf ("Received request: [%s]\n", $request);

    // 做一些“处理”
    sleep (1);

    // 应答World
    $responder->send("World");
}
```

下面是客户端的代码：

hwclient: Hello World client in C

```c
//
//  Hello World 客户端
//  连接REQ套接字至 tcp://localhost:5555
//  发送Hello给服务端，并接收World
//
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main (void)
{
    void *context = zmq_init (1);

    //  连接至服务端的套接字
    printf ("正在连接至hello world服务端...\n");
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        zmq_msg_t request;
        zmq_msg_init_size (&request, 5);
        memcpy (zmq_msg_data (&request), "Hello", 5);
        printf ("正在发送 Hello %d...\n", request_nbr);
        zmq_send (requester, &request, 0);
        zmq_msg_close (&request);

        zmq_msg_t reply;
        zmq_msg_init (&reply);
        zmq_recv (requester, &reply, 0);
        printf ("接收到 World %d\n", request_nbr);
        zmq_msg_close (&reply);
    }
    zmq_close (requester);
    zmq_term (context);
    return 0;
}
```

这看起来是否太简单了？ZMQ就是这样一个东西，你往里加点儿料就能制作出一枚无穷能量的原子弹，用它来拯救世界吧！

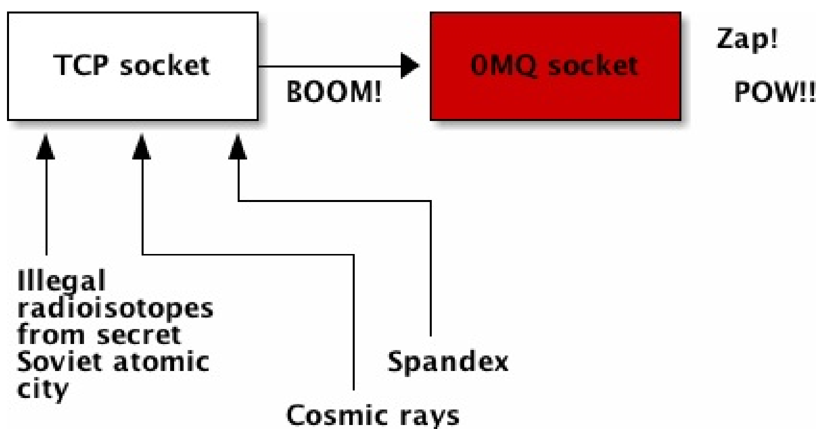


Figure 2 — A terrible accident...

理论上你可以连接千万个客户端到这个服务端上，同时连接都没问题，程序仍会运作得很好。你可以尝试一下先打开客户端，再打开服务端，可以看到程序仍然会正常工作，想想这意味着什么。

让我简单介绍一下这两段程序到底做了什么。首先，他们创建了一个ZMQ上下文，然后是一个套接字。不要被这些陌生的名词吓到，后面我们都会讲到。服务端将REP套接字绑定到5555端口上，并开始等待请求，发出应答，如此循环。客户端则是发送请求并等待服务端的应答。

这些代码背后其实发生了很多很多事情，但是程序员完全不必理会这些，只要知道这些代码短小精悍，极少出错，耐高压。这种通信模式我们称之为请求-应答模式，是ZMQ最直接的一种应用。你可以拿它和RPC及经典的C/S模型做类比。

关于字符串

ZMQ不会关心发送消息的内容，只要知道它所包含的字节数。所以，程序员需要做一些工作，保证对方节点能够正确读取这些消息。如何将一个对象或复杂数据类型转换成ZMQ可以发送的消息，这有类似Protocol Buffers的序列化软件可以做到。但对于字符串，你也是需要有所注意的。

在C语言中，字符串都以一个空字符结尾，你可以像这样发送一个完整的字符串：

```
zmq_msg_init_data (&request, "Hello", 6, NULL, NULL);
```

但是，如果你用其他语言发送这个字符串，很可能不会包含这个空字节，如你使用Python发送：

```
socket.send ("Hello")
```

实际发送的消息是：

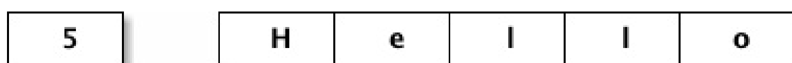


Figure 3 — A 0MQ string

如果你从C语言中读取该消息，你会读到一个类似于字符串的内容，甚至它可能就是一个字符串（第六位在内存中正好是一个空字符），但是这并不合适。这样一来，客户端和服务端对字符串的定义就不统一了，你会得到一些奇怪的结果。

当你用C语言从ZMQ中获取字符串，你不能够相信该字符串有一个正确的结尾。因此，当你在接受字符串时，应该建立多一个字节的缓冲区，将字符串放进去，并添加结尾。

所以，让我们做如下假设：**ZMQ**的字符串是有长度的，且传送时不加结束符。在最简单的情况下，ZMQ字符串和ZMQ消息中的一帧是等价的，就如上图所展现的，由一个长度属性和一串字节表示。

下面这个功能函数会帮助我们在C语言中正确的接受字符串消息：

```
// 从ZMQ套接字中接收字符串，并转换为C语言的字符串
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_recv (socket, &message, 0);
    int size = zmq_msg_size (&message);
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}
```

这段代码我们会在日后的示例中使用，我们可以顺手写一个s_send()方法，并打包成一个.h文件供我们使用。

这就诞生了zhelpers.h，一个供C语言使用的ZMQ功能函数库。它的源代码比较长，而且只对C语言程序员有用，你可以在闲暇时看一看。

获取版本号

ZMQ目前有多个版本，而且仍在持续更新。如果你遇到了问题，也许这在下一个版本中已经解决了。想知道目前的ZMQ版本，你可以在程序中运行如下：

version: ØMQ version reporting in C

```
//
// 返回当前ZMQ的版本号
//
#include "zhelpers.h"

int main (void)
{
    int major, minor, patch;
    zmq_version (&major, &minor, &patch);
    printf ("当前ZMQ版本号为 %d.%d.%d\n", major, minor, patch);

    return EXIT_SUCCESS;
}
```

让消息流动起来

第二种经典的消息模式是单向数据分发：服务端将更新事件发送给一组客户端。让我们看一个天气信息发布的例子，包括邮编、温度、相对湿度。我们生成这些随机信息，用来模拟气象站所做的那样。

下面是服务端的代码，使用5556端口：

wuserver: Weather update server in C

```
//
//  气象信息更新服务
//  绑定PUB套接字至tcp://*:5556端点
//  发布随机气象信息
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和PUB套接字
    void *context = zmq_init (1);
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5556");
    zmq_bind (publisher, "ipc://weather.ipc");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));
    while (1) {
        //  生成数据
        int zipcode, temperature, relhumidity;
        zipcode     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

        //  向所有订阅者发送消息
        char update [20];
        sprintf (update, "%05d %d %d", zipcode, temperature, relhur
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

这项更新服务没有开始、没有结束，就像永不消失的电波一样。

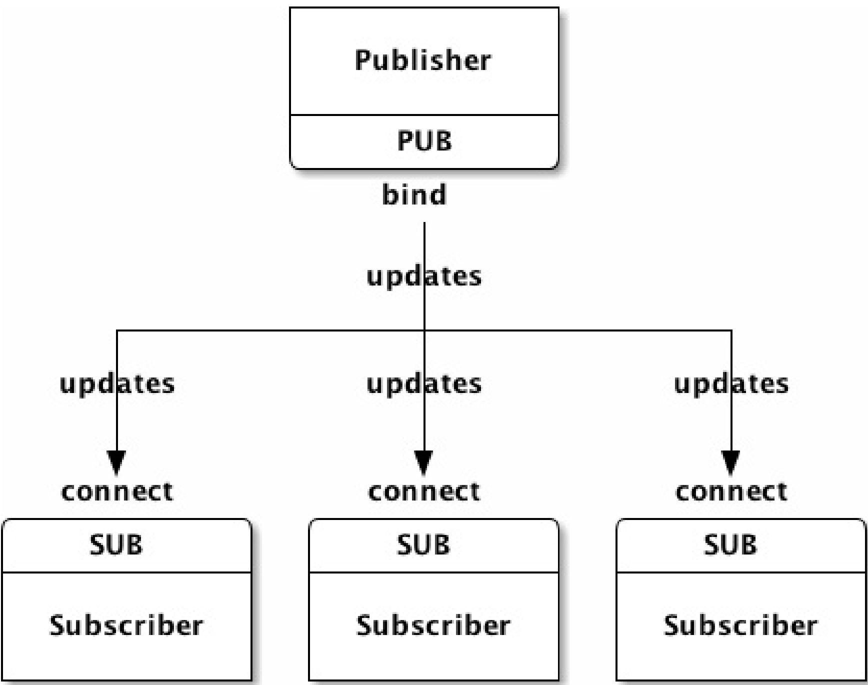


Figure 4 — Publish-Subscribe

下面是客户端程序，它会接受发布者的消息，只处理特定邮编标注的信息，如纽约的邮编是10001:

wuclient: Weather update client in C

```
//
//  气象信息客户端
//  连接SUB套接字至tcp://*:5556端点
//  收集指定邮编的气象信息，并计算平均温度
//
#include "zhelpers.h"

int main (int argc, char *argv [])
{
    void *context = zmq_init (1);

    //  创建连接至服务端的套接字
    printf ("正在收集气象信息...\n");
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");

    //  设置订阅信息，默认为纽约，邮编10001
    char *filter = (argc > 1)? argv [1]: "10001 ";
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, filter, strlen (filt

    //  处理100条更新信息
    int update_nbr;
    long total_temp = 0;
    for (update_nbr = 0; update_nbr < 100; update_nbr++) {
        char *string = s_recv (subscriber);
        int zipcode, temperature, relhumidity;
        sscanf (string, "%d %d %d",
            &zipcode, &temperature, &relhumidity);
        total_temp += temperature;
        free (string);
    }
    printf ("地区邮编 '%s' 的平均温度为 %dF\n",
        filter, (int) (total_temp / update_nbr));

    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

需要注意的是，在使用SUB套接字时，必须使用zmq_setsockopt()方法来设置订阅的内容。如果你不设置订阅内容，那将什么消息都收不到，新手很容易犯这个错误。订阅信息可以是任何字符串，可以设置多次。只要消息满足其中一条订阅信息，SUB套接字就会收到。订阅者可以选择不接收某类消息，也是通过zmq_setsockopt()方法实现的。

PUB-SUB套接字组合是异步的。客户端在一个循环体中使用zmq_recv()接收消息，如果向SUB套接字发送消息则会报错；类似地，服务端可以不断地使用zmq_send()发送消息，但不能在PUB套接字上使用zmq_recv()。

关于PUB-SUB套接字，还有一点需要注意：你无法得知SUB是何时开始接收消息的。就算你先打开了SUB套接字，后打开PUB发送消息，这时SUB还是会丢失一些消息的，因为建立连接是需要一些时间的。很少，但并不是零。

这种“慢连接”的症状一开始会让很多人困惑，所以这里我要详细解释一下。还记得ZMQ是在后台进行异步的I/O传输的，如果你有两个节点用以下顺序相连：

- 订阅者连接至端点接收消息并计数；
- 发布者绑定至端点并立刻发送1000条消息。

运行的结果很可能是订阅者一条消息都收不到。这时你可能会傻眼，忙于检查有没有设置订阅信息，并重新尝试，但结果还是一样。

我们知道在建立TCP连接时需要进行三次握手，会耗费几毫秒的时间，而当节点数增加时这个数字也会上升。在这么短的时间里，ZMQ就可以发送很多很多消息了。举例来说，如果建立连接需要耗时5毫秒，而ZMQ只需要1毫秒就可以发送完这1000条消息。

第二章中我会解释如何使发布者和订阅者同步，只有当订阅者准备好时发布者才会开始发送消息。有一种简单的方法来同步PUB和SUB，就是让PUB延迟一段时间再发送消息。现实编程中我不建议使用这种方式，因为它太脆弱了，而且不好控制。不过这里我们先暂且使用sleep的方式来解决，等到第二章的时候再讲述正确的处理方式。

另一种同步的方式则是认为发布者的消息流是无穷无尽的，因此丢失了前面一部分信息也没有关系。我们的气象信息客户端就是这么做的。

示例中的气象信息客户端会收集指定邮编的一千条信息，其间大约有1000万条信息被发布。你可以先打开客户端，再打开服务端，工作一段时间后重启服务端，这时客户端仍会正常工作。当客户端收集完所需信息后，会计算并输出平均温度。

关于发布-订阅模式的几点说明：

- 订阅者可以连接多个发布者，轮流接收消息；
- 如果发布者没有订阅者与之相连，那它发送的消息将直接被丢弃；
- 如果你使用TCP协议，那当订阅者处理速度过慢时，消息会在发布者处堆积。以后我们会讨论如何使用阈值（HWM）来保护发布者。
- 在目前版本的ZMQ中，消息的过滤是在订阅者处进行的。也就是说，发布者会向订阅者发送所有的消息，订阅者会将未订阅的消息丢弃。

我在自己的四核计算机上尝试发布1000万条消息，速度很快，但没什么特别的：

```
ph@ws200901:~/work/git/0MQGuide/examples/c$ time wuclient
Collecting updates from weather server...
Average temperature for zipcode '10001 ' was 18F

real    0m5.939s
user    0m1.590s
sys     0m2.290s
```

分布式处理

下面一个示例程序中，我们将使用ZMQ进行超级计算，也就是并行处理模型：

- 任务分发器会生成大量可以并行计算的任务；
- 有一组worker会处理这些任务；
- 结果收集器会在末端接收所有worker的处理结果，进行汇总。

现实中，worker可能散落在不同的计算机中，利用GPU（图像处理单元）进行复杂计算。下面是任务分发器的代码，它会生成100个任务，任务内容是让收到的worker延迟若干毫秒。

taskvent: Parallel task ventilator in C

```
//
//  任务分发器
//  绑定PUSH套接字至tcp://localhost:5557端点
//  发送一组任务给已建立连接的worker
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  用于发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_bind (sender, "tcp://*:5557");

    //  用于发送开始信号的套接字
    void *sink = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sink, "tcp://localhost:5558");

    printf ("准备好worker后按任意键开始: ");
    getchar ();
    printf ("正在向worker分配任务...\n");

    //  发送开始信号
    s_send (sink, "0");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));

    //  发送100个任务
    int task_nbr;
    int total_msec = 0;     //  预计执行时间（毫秒）
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  随机产生1-100毫秒的工作量
        workload = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("预计执行时间: %d 毫秒\n", total_msec);
    sleep (1);              //  延迟一段时间，让任务分发完成

    zmq_close (sink);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

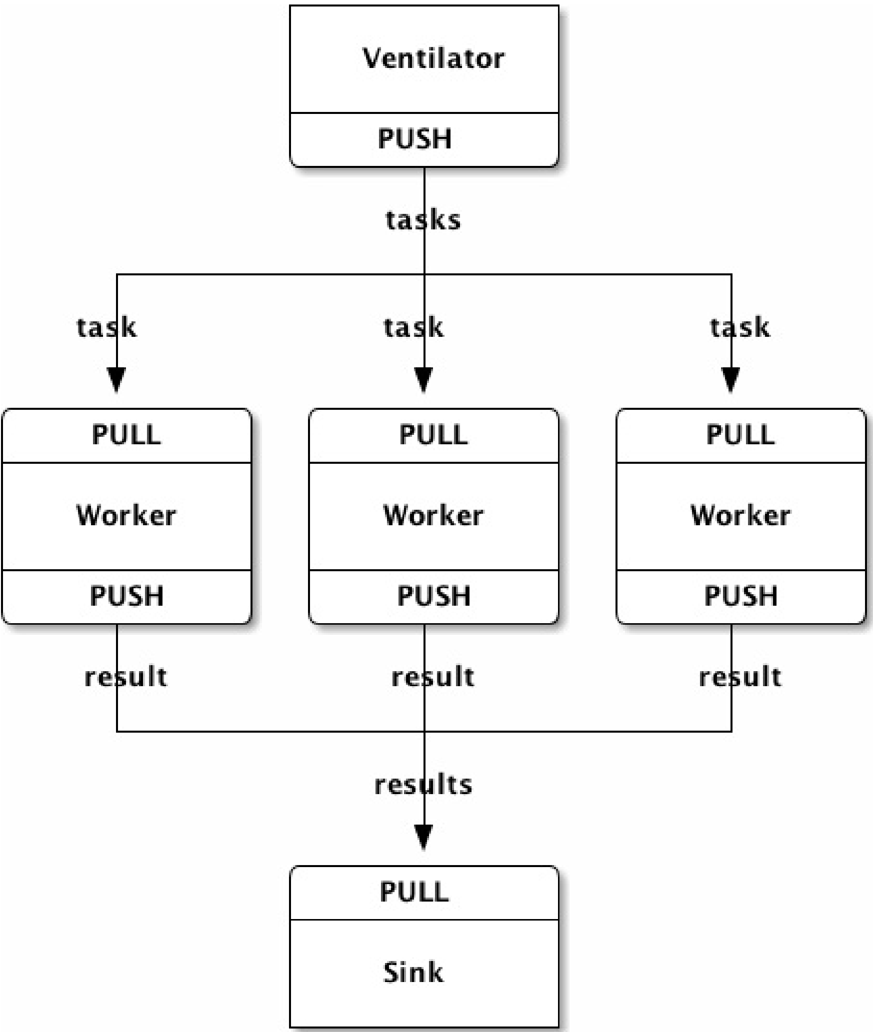


Figure 5 — Parallel Pipeline

下面是worker的代码，它接受信息并延迟指定的毫秒数，并发送执行完毕的信号：

taskwork: Parallel task worker in C

```
//
//  任务执行器
//  连接PULL套接字至tcp://localhost:5557端点
//  从任务分发器处获取任务
//  连接PUSH套接字至tcp://localhost:5558端点
//  向结果采集器发送结果
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  获取任务的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  发送结果的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  循环处理任务
    while (1) {
        char *string = s_recv (receiver);
        //  输出处理进度
        fflush (stdout);
        printf ("%s.", string);

        //  开始处理
        s_sleep (atoi (string));
        free (string);

        //  发送结果
        s_send (sender, "");
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

下面是结果收集器的代码。它会收集100个处理结果，并计算总的执行时间，让我们由此判别任务是否是并行计算的。

tasksink: Parallel task sink in C

```
//
//  任务收集器
//  绑定PULL套接字至tcp://localhost:5558端点
//  从worker处收集处理结果
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    //  等待开始信号
    char *string = s_recv (receiver);
    free (string);

    //  开始计时
    int64_t start_time = s_clock ();

    //  确定100个任务均已处理
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    //  计算并输出总执行时间
    printf ("执行时间: %d 毫秒\n",
        (int) (s_clock () - start_time));

    zmq_close (receiver);
    zmq_term (context);
    return 0;
}
```

一组任务的平均执行时间在5秒左右，以下是分别开始1个、2个、4个worker时的执行结果：

```
#  1 worker
Total elapsed time: 5034 msec
#  2 workers
Total elapsed time: 2421 msec
#  4 workers
Total elapsed time: 1018 msec
```

关于这段代码的几个细节：

- worker上游和任务分发器相连，下游和结果收集器相连，这就意味着你可以开启任意多个worker。但若worker是绑定至端点的，而非连接至端点，那我们就需要准备更多的端点，并配置任务分发器和结果收集器。所以说，任务分发器和结果收集器是这个网络结构中较为稳定的部分，因此应该由它们绑定至端点，而非worker，因为它们较为动态。

- 我们需要做一些同步的工作，等待worker全部启动之后再分发任务。这点在ZMQ中很重要，且不易解决。连接套接字的动作会耗费一定的时间，因此当第一个worker连接成功时，它会一下收到很多任务。所以说，如果我们不进行同步，那这些任务根本就不会被并行地执行。你可以自己试验一下。

- 任务分发器使用PUSH套接字向worker均匀地分发任务（假设所有的worker都已经连接上了），这种机制称为负载均衡，以后我们会见得更多。

- 结果收集器的PULL套接字会均匀地从worker处收集消息，这种机制称为公平队列：

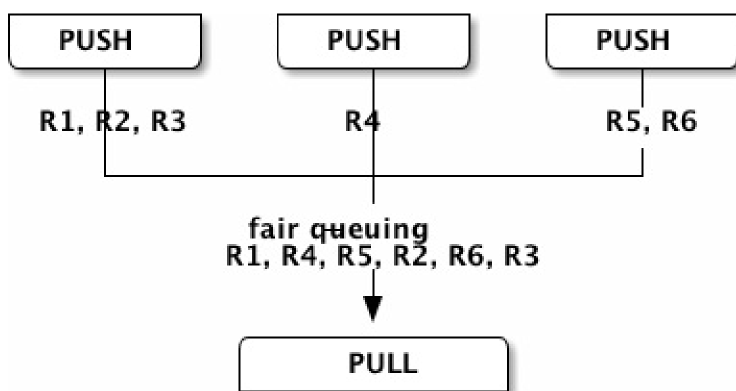


Figure 6 — Fair queuing

管道模式也会出现慢连接的情况，让人误以为PUSH套接字没有进行负载均衡。如果你的程序中某个worker接收到了更多的请求，那是因为它的PULL套接字连接得比较快，从而在别的worker连接之前获取了额外的消息。

使用ZMQ编程

看着这些示例程序后，你一定迫不及待想要用ZMQ进行编程了。不过在开始之前，我还有几条建议想给到你，这样可以省去未来的一些麻烦：

- 学习ZMQ要循序渐进，虽然它只是一套API，但却提供了无尽的可能。一步一步学习它提供的功能，并完全掌握。

- 编写漂亮的代码。丑陋的代码会隐藏问题，让想要帮助你的人无从下手。比如，你会习惯于使用无意义的变量名，但读你代码的人并不知道。应使用有意义的变量名称，而不是随意起一个。代码的缩进要统一，布局清晰。漂亮的代码可以让你的世界变得更美好。

- 边写边测试，当代码出现问题，你就可以快速定位到某些行。这一点在编写ZMQ应用程序时尤为重要，因为很多时候你无法第一次就编写出正确的代码。

- 当你发现自己编写的代码无法正常工作时，你可以将其拆分成一些代码片段，看看哪段没有正确地执行。ZMQ可以让你构建非常模块化的代码，所以应该好好利用这一点。
- 需要时应使用抽象的方法来编写程序（类、成员函数等等），不要随意拷贝代码，因为拷贝代码的同时也是在拷贝错误。

我们看看下面这段代码，是某位同仁让我帮忙修改的：

```
//  注意：不要使用这段代码！
static char *topic_str = "msg.x|";

void* pub_worker(void* arg){
    void *ctx = arg;
    assert(ctx);

    void *qskt = zmq_socket(ctx, ZMQ_REP);
    assert(qskt);

    int rc = zmq_connect(qskt, "inproc://querys");
    assert(rc == 0);

    void *pubskt = zmq_socket(ctx, ZMQ_PUB);
    assert(pubskt);

    rc = zmq_bind(pubskt, "inproc://publish");
    assert(rc == 0);

    uint8_t cmd;
    uint32_t nb;
    zmq_msg_t topic_msg, cmd_msg, nb_msg, resp_msg;

    zmq_msg_init_data(&topic_msg, topic_str, strlen(topic_str) , NU

    fprintf(stdout,"WORKER: ready to recieve messages\n");
    //  注意：不要使用这段代码，它不能工作！
    //  e.g. topic_msg will be invalid the second time through
    while (1){
    zmq_send(pubskt, &topic_msg, ZMQ_SNDMORE);

    zmq_msg_init(&cmd_msg);
    zmq_recv(qskt, &cmd_msg, 0);
    memcpy(&cmd, zmq_msg_data(&cmd_msg), sizeof(uint8_t));
    zmq_send(pubskt, &cmd_msg, ZMQ_SNDMORE);
    zmq_msg_close(&cmd_msg);

    fprintf(stdout, "recieved cmd %u\n", cmd);

    zmq_msg_init(&nb_msg);
    zmq_recv(qskt, &nb_msg, 0);
    memcpy(&nb, zmq_msg_data(&nb_msg), sizeof(uint32_t));
    zmq_send(pubskt, &nb_msg, 0);
```

```
        zmq_msg_close(&nb_msg);

        fprintf(stdout, "recieved nb %u\n", nb);

        zmq_msg_init_size(&resp_msg, sizeof(uint8_t));
        memset(zmq_msg_data(&resp_msg), 0, sizeof(uint8_t));
        zmq_send(qskt, &resp_msg, 0);
        zmq_msg_close(&resp_msg);

    }
    return NULL;
}
```

下面是我为他重写的代码，顺便修复了一些BUG：

```
static void *
worker_thread (void *arg) {
    void *context = arg;
    void *worker = zmq_socket (context, ZMQ_REP);
    assert (worker);
    int rc;
    rc = zmq_connect (worker, "ipc://worker");
    assert (rc == 0);

    void *broadcast = zmq_socket (context, ZMQ_PUB);
    assert (broadcast);
    rc = zmq_bind (broadcast, "ipc://publish");
    assert (rc == 0);

    while (1) {
        char *part1 = s_recv (worker);
        char *part2 = s_recv (worker);
        printf ("Worker got [%s][%s]\n", part1, part2);
        s_sendmore (broadcast, "msg");
        s_sendmore (broadcast, part1);
        s_send (broadcast, part2);
        free (part1);
        free (part2);

        s_send (worker, "OK");
    }
    return NULL;
}
```

上段程序的最后，它将套接字在两个线程之间传递，这会导致莫名其妙的问题。这种行为在ZMQ 2.1中虽然是合法的，但是不提倡使用。

ZMQ 2.1版

历史告诉我们，ZMQ 2.0是一个低延迟的分布式消息系统，它从众多同类软件中脱颖而出，摆脱了各种奢华的名目，向世界宣告“无极限”的口号。这是我们一直在使用的稳定发行版。

时过境迁，2010年流行的东西在2011年就不一定了。当ZMQ的开发者和社区开发者在激烈地讨论ZMQ的种种问题时，ZMQ 2.1横空出世了，成为新的稳定发行版。

本指南主要针对ZMQ 2.1进行描述，因此对于从ZMQ 2.0迁移过来的开发者来说有一些需要注意的地方：

- 在2.0中，调用zmq_close()和zmq_term()时会丢弃所有尚未发送的消息，所以在发送完消息后不能直接关闭程序，2.0的示例中往往使用sleep(1)来规避这个问题。但是在2.1中就不需要这样做了，程序会等待消息全部发送完毕后再退出。

- 相反地，2.0中可以在尚有套接字打开的情况下调用zmq*term()*，这在*2.1*中会变得不安全，会造成程序的阻塞。所以，在*2.1*程序中我们会先关闭所有的套接字，然后才退出程序。如果套接字中有尚未发送的消息，程序就会一直处于等待状态，除非手工设置了套接字的LINGER选项_（如设置为零），那么套接字会在相应的时间后关闭。

```
int zero = 0;
zmq_setsockopt (mysocket, ZMQ_LINGER, &zero, sizeof (zero));
```

- 2.0中，zmq_poll()函数没有定时功能，它会在满足条件时立刻返回，我们需要在循环体中检查还有多少剩余。但在2.1中，zmq_poll()会在指定时间后返回，因此可以作为定时器使用。

- 2.0中，ZMQ会忽略系统的中断消息，这就意味着对libzmq的调用是不会收到EINTR消息的，这样就无法对SIGINT（Ctrl-C）等消息进行处理了。在2.1中，这个问题得以解决，像类似zmq_recv()的方法都会接收并返回系统的EINTR消息。

正确地使用上下文

ZMQ应用程序的一开始总是会先创建一个上下文，并用它来创建套接字。在C语言中，创建上下文的函数是zmq_init()。一个进程中只应该创建一个上下文。从技术的角度来说，上下文是一个容器，包含了该进程下所有的套接字，并为inproc协议提供实现，用以高速连接进程内不同的线程。如果一个进程中创建了两个上下文，那就相当于启动了两个ZMQ实例。如果这正是你需要的，那没有问题，但一般情况下：

在一个进程中使用**zmq_init()**函数创建一个上下文，并在结束时使用**zmq_term()**函数关闭它

如果你使用了fork()系统调用，那每个进程需要自己的上下文对象。如果在调用fork()之前调用了zmq_init()函数，那每个子进程都会有自己的上下文对象。通常情况下，你会需要在子进程中做些有趣的事，而让父进程来管理它们。

正确地退出和清理

程序员的一个良好习惯是：总是在结束时进行清理工作。当你使用像Python那样的语言编写ZMQ应用程序时，系统会自动帮你完成清理。但如果使用的是C语言，那就需要小心地处理了，否则可能发生内存泄露、应用程序不稳定等问题。

内存泄露只是问题之一，其实ZMQ是很在意程序的退出方式的。个中原因比较复杂，但简单的来说，如果仍有套接字处于打开状态，调用zmq_term()时会导致程序挂起；就算关闭了所有的套接字，如果仍有消息处于待发送状态，zmq_term()也会造成程序的等待。只有当套接字的LINGER选项设为0时才能避免。

我们需要关注的ZMQ对象包括：消息、套接字、上下文。好在内容并不多，至少在一般的应用程序中是这样：

- 处理完消息后，记得用zmq_msg_close()函数关闭消息；
- 如果你同时打开或关闭了很多套接字，那可能需要重新规划一下程序的结构了；
- 退出程序时，应该先关闭所有的套接字，最后调用zmq_term()函数，销毁上下文对象。

如果要用ZMQ进行多线程的编程，需要考虑的问题就更多了。我们会在下一章中详述多线程编程，但如果你耐不住性子想要尝试一下，以下是在退出时的一些建议：

- 不要在多个线程中使用同一个套接字。不要去想为什么，反正别这么干就是了。
- 关闭所有的套接字，并在主程序中关闭上下文对象。
- 如果仍有处于阻塞状态的recv或poll调用，应该在主程序中捕捉这些错误，并在相应的线程中关闭套接字。不要重复关闭上下文，zmq_term()函数会等待所有的套接字安全地关闭后才结束。

看吧，过程是复杂的，所以不同语言的API实现者可能会将这些步骤封装起来，让结束程序变得不那么复杂。

我们为什么需要ZMQ

现在我们已经将ZMQ运行起来了，让我们回顾一下为什么我们需要ZMQ：

目前的应用程序很多都会包含跨网络的组件，无论是局域网还是因特网。这些程序的开发者都会用到某种消息通信机制。有些人会使用某种消息队列产品，而大多数人则会自己手工来做这些事，使用TCP或UDP协议。这些协议使用起来并不困难，但是，简单地将消息从A发给B，和在任何情况下都能进行可靠的消息传输，这两种情况显然是不同的。

让我们看看在使用纯TCP协议进行消息传输时会遇到的一些典型问题。任何可复用的消息传输层肯定或多或少地会要解决以下问题：

- 如何处理I/O？是让程序阻塞等待响应，还是在后台处理这些事？这是软件设计的关键因素。阻塞式的I/O操作会让程序架构难以扩展，而后台处理I/O也是比较困难的。

- 如何处理那些临时的、来去自由的组件？我们是否要将组件分为客户端和服务端两种，并要求服务端永不消失？那如果我们想要将服务端相连怎么办？我们要每隔几秒就进行重连吗？
- 我们如何表示一条消息？我们怎样通过拆分消息，让其变得易读易写，不用担心缓存溢出，既能高效地传输小消息，又能胜任视频等大型文件的传输？
- 如何处理那些不能立刻发送出去的消息？比如我们需要等待一个网络组件重新连接的时候？我们是直接丢弃该条消息，还是将它存入数据库，或是内存中的一个队列？
- 要在哪里保存消息队列？如果某个组件读取消息队列的速度很慢，造成消息的堆积怎么办？我们要采取什么样的策略？
- 如何处理丢失的消息？我们是等待新的数据，请求重发，还是需要建立一套新的可靠性机制以保证消息不会丢失？如果这个机制自身崩溃了呢？
- 如果我们想换一种网络连接协议，如用广播代替TCP单播？或者改用IPv6？我们是否需要重写所有的应用程序，或者将这种协议抽象到一个单独的层中？
- 我们如何对消息进行路由？我们可以将消息同时发送给多个节点吗？是否能将应答消息返回给请求的发送方？
- 我们如何为另一种语言写一个API？我们是否需要完全重写某项协议，还是重新打包一个类库？
- 怎样才能做到在不同的架构之间传送消息？是否需要为消息规定一种编码？
- 我们如何处理网络通信错误？等待并重试，还是直接忽略或取消？

我们可以找一个开源软件来做例子，如Hadoop Zookeeper，看一下它的C语言API源码，src/c/src/zookeeper.c。这段代码大约有3200行，没有注释，实现了一个C/S网络通信协议。它工作起来很高效，因为使用了poll()来代替select()。但是，Zookeeper应该被抽象出来，作为一种通用的消息通信层，并加以详细的注释。像这样的模块应该得到最大程度上的复用，而不是重复地制造轮子。

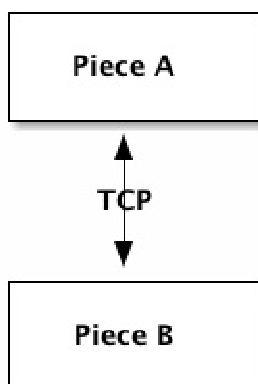


Figure 7 — Messaging as it starts

但是，如何编写这样一个可复用的消息层呢？为什么长久以来人们宁愿在自己的代码中重复书写控制原始TCP套接字的代码，而不愿编写这样一个公共库呢？

其实，要编写一个通用的消息层是件非常困难的事，这也是为什么FOSS项目不断在尝试，一些商业化的消息产品如此之复杂、昂贵、僵硬、脆弱。2006年，iMatix设计了AMQP协议，为FOSS项目的开发者提供了可能是当时第一个可复用的消息系统。AMQP比其他同类产品要来得好，但仍然是复杂、昂贵和脆弱的。它需要花费几周的时间去学习，花费数月的时间去创建一个真正能用的架构，到那时可能为时已晚了。

大多数消息系统项目，如AMQP，为了解决上面提到的种种问题，发明了一些新的概念，如“代理”的概念，将寻址、路由、队列等功能都包含了进来。结果就是在一个没有任何注释的协议之上，又构建了一个C/S协议和相应的API，让应用程序和代理相互通信。代理的确是一个不错的解决方案，帮助降低大型网络结构的复杂度。但是，在Zookeeper这样的项目中应用代理机制的消息系统，可能是件更加糟糕的事，因为这意味了需要添加一台新的计算机，并构成一个新的单点故障。代理会逐渐成为新的瓶颈，管理起来更具风险。如果软件支持，我们可以添加第二个、第三个、第四个代理，构成某种冗余容错的模式。有人就是这么做的，这让系统架构变得更为复杂，增加了隐患。

在这种以代理为中心的架构下，需要一支专门的运维团队。你需要昼夜不停地观察代理的状态，不时地用棍棒调教他们。你需要添加计算机，以及更多的备份机，你需要有专人管理这些机器。这样做只对那些大型的网络应用程序才有意义，因为他们有更多可移动的模块，有多个团队进行开发和维护，而且已经经过了多年的建设。

这样一来，中小应用程序的开发者们就无计可施了。他们只能设法避免编写网络应用程序，转而编写那些不需要扩展的程序；或者可以使用原始的方式进行网络编程，但编写的软件会非常脆弱和复杂，难以维护；亦或者他们选择一种消息通信产品，虽然能够开发出扩展性强的应用程序，但需要支付高昂的代价。似乎没有一种选择是合理的，这也是为什么在上个世纪消息系统会成为一个广泛的问题。

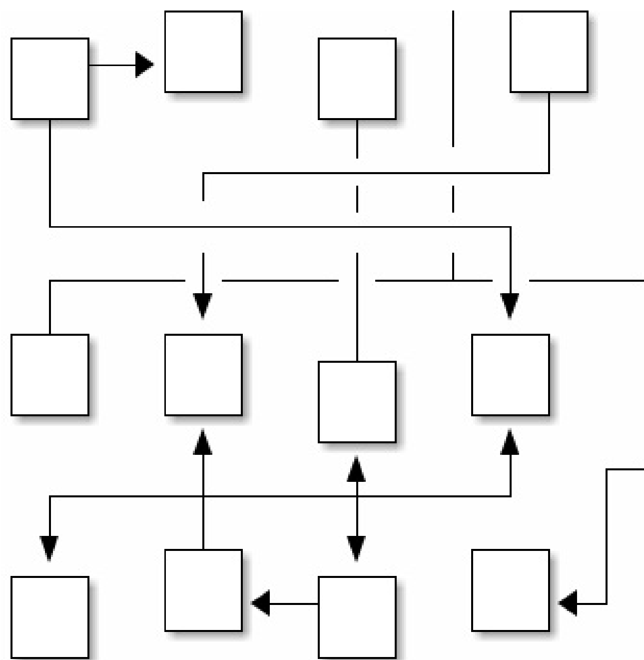


Figure 8 — Messaging as it becomes

我们真正需要的是这样一种消息软件，它能够做大型消息软件所能做的一切，但使用起来又非常简单，成本很低，可以用到所有的应用程序中，没有任何依赖条件。因为没有了额外的模块，就降低了出错的概率。这种软件需要能够在所有的操作系统上运行，并能支持所有的编程语言。

ZMQ就是这样一种软件：它高效，提供了嵌入式的类库，使应用程序能够很好地在网络中扩展，成本低廉。

ZMQ的主要特点有：

- ZMQ会在后台线程异步地处理I/O操作，它使用一种不会死锁的数据结构来存储消息。
- 网络组件可以来去自如，ZMQ会负责自动重连，这就意味着你可以以任何顺序启动组件；用它创建的面向服务架构（SOA）中，服务端可以随意地加入或退出网络。
- ZMQ会在有必要的情况下自动将消息放入队列中保存，一旦建立了连接就开始发送。
- ZMQ有阈值（HWM）的机制，可以避免消息溢出。当队列已满，ZMQ会自动阻塞发送者，或丢弃部分消息，这些行为取决于你所使用的消息模式。
- ZMQ可以让你用不同的通信协议进行连接，如TCP、广播、进程内、进程间。改变通信协议时你不需要去修改代码。
- ZMQ会恰当地处理速度较慢的节点，会根据消息模式使用不同的策略。
- ZMQ提供了多种模式进行消息路由，如请求-应答模式、发布-订阅模式等。这些模式可以用来搭建网络拓扑结构。
- ZMQ中可以根据消息模式建立起一些中间装置（很小巧），可以用来降低网络的复杂程度。
- ZMQ会发送整个消息，使用消息帧的机制来传递。如果你发送了10KB大小的消息，你就会收到10KB大小的消息。
- ZMQ不强制使用某种消息格式，消息可以是0字节的，或是大到GB级的数据。当你表示这些消息时，可以选用诸如谷歌的protocol buffers，XDR等序列化产品。
- ZMQ能够智能地处理网络错误，有时它会进行重试，有时会告知你某项操作发生了错误。
- ZMQ甚至可以降低对环境的污染，因为节省了CPU时间意味着节省了电能。

其实ZMQ可以做的还不止这些，它会颠覆人们编写网络应用程序的模式。虽然从表面上看，它不过是提供了一套处理套接字的API，能够用zmq_recv()和zmq_send()进行消息的收发，但是，消息处理将成为应用程序的核心部分，很快你的程序就会变成一个个消息处理模块，这既美观又自然。它的扩展性还很强，每项任务由一个节点（节点是一个线程）、同一台机器上的两个节点（节点是一个进程）、同一网络上的两台机器（节点是一台机器）来处理，而不需要改动应用程序。

套接字的扩展性

我们来用实例看看ZMQ套接字的扩展性。这个脚本会启动气象信息服务及多个客户端：

```
wuserver &
wuclient 12345 &
wuclient 23456 &
wuclient 34567 &
wuclient 45678 &
wuclient 56789 &
```

执行过程中，我们可以通过top命令查看进程状态（以下是一台四核机器的情况）：

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAN
 7136 ph        20   0 1040m 959m 1156 R  157 12.0  16:25.47 wuserv
 7966 ph        20   0 98608 1804 1372 S   33  0.0   0:03.94 wuclie
 7963 ph        20   0 33116 1748 1372 S   14  0.0   0:00.76 wuclie
 7965 ph        20   0 33116 1784 1372 S    6  0.0   0:00.47 wuclie
 7964 ph        20   0 33116 1788 1372 S    5  0.0   0:00.25 wuclie
 7967 ph        20   0 33072 1740 1372 S    5  0.0   0:00.35 wuclie
```

我们想想现在发生了什么：气象信息服务程序有一个单独的套接字，却能同时向五个客户端并行地发送消息。我们可以有成百上千个客户端并行地运作，服务端看不到这些客户端，不能操纵它们。

如果解决丢失消息的问题

在编写ZMQ应用程序时，你遇到最多的问题可能是无法获得消息。下面有一个问题解决路线图，列举了最基本的出错原因。不用担心其中的某些术语你没有见过，在后面的几章里都会讲到。

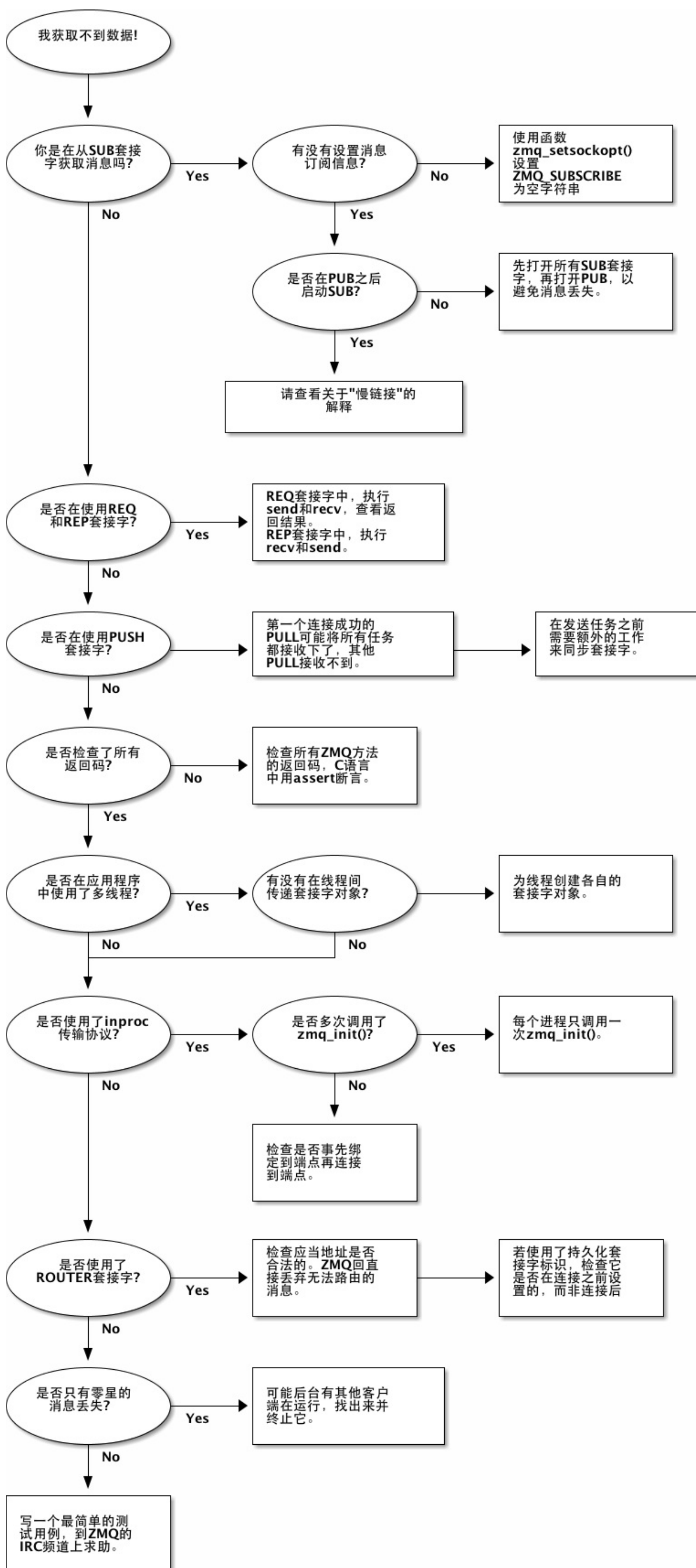


Figure 9 — Missing Message Problem Solver

如果ZMQ在你的应用程序中扮演非常重要的角色，那你可能就需要好好计划一下了。首先，创建一个原型，用以测试设计方案的可行性。采取一些压力测试的手段，确保它足够的健壮。其次，主攻测试代码，也就是编写测试框架，保证有足够的电力供应和时间，来进行高强度的测试。理想状态下，应该由一个团队编写程序，另一个团队负责击垮它。最后，让你的公司及时联系iMatix，获得技术上的支持。

简而言之，如果你没有足够理由说明设计出来的架构能够在现实环境中运行，那么很有可能它就会在最紧要的关头崩溃。

警告：你的想法可能会被颠覆！

传统网络编程的一个规则是套接字只能和一个节点建立连接。虽然也有广播的协议，但毕竟是第三方的。当我们认定“一个套接字 = 一个连接”的时候，我们会用一些特定的方式来扩展应用程序架构：我们为每一块逻辑创建线程，该线程独立地维护一个套接字。

但在ZMQ的世界里，套接字是智能的、多线程的，能够自动地维护一组完整的连接。你无法看到它们，甚至不能直接操纵这些连接。当你进行消息的收发、轮询等操作时，只能和ZMQ套接字打交道，而不是连接本身。所以说，ZMQ世界里的连接是私有的，不对外部开放，这也是ZMQ易于扩展的原因之一。

由于你的代码只会和某个套接字进行通信，这样就可以处理任意多个连接，使用任意一种网络协议。而ZMQ的消息模式又可以进行更为廉价和便捷的扩展。

这样一来，传统的思维就无法在ZMQ的世界里应用了。在你阅读示例程序代码的时候，也许你脑子里会想方设法地将这些代码和传统的网络编程相关联：当你读到“套接字”的时候，会认为它就表示与另一个节点的连接——这种想法是错误的；当你读到“线程”时，会认为它是与另一个节点的连接——这也是错误的。

如果你是第一次阅读本指南，使用ZMQ进行了一两天的开发（或者更长），可能会觉得疑惑，ZMQ怎么会让事情便得如此简单。你再次尝试用以往的思维去理解ZMQ，但又无功而返。最后，你会被ZMQ的理念所折服，拨云见雾，开始享受ZMQ带来的乐趣。

第二章 ZeroMQ进阶

第一章我们简单试用了ZMQ的若干通信模式：请求-应答模式、发布-订阅模式、管道模式。这一章我们将学习更多在实际开发中会使用到的东西：

本章涉及的内容有：

- 创建和使用ZMQ套接字
- 使用套接字发送和接收消息
- 使用ZMQ提供的异步I/O套接字构建你的应用程序
- 在单一线程中使用多个套接字
- 恰当地处理致命和非致命错误
- 处理诸如Ctrl-C的中断信号
- 正确地关闭ZMQ应用程序
- 检查ZMQ应用程序的内存泄露
- 发送和接收多帧消息
- 在网络中转发消息
- 建立简单的消息队列代理
- 使用ZMQ编写多线程应用程序
- 使用ZMQ在线程间传递信号
- 使用ZMQ协调网络中的节点
- 使用标识创建持久化套接字
- 在发布-订阅模式中创建和使用消息信封
- 如何让持久化的订阅者能够从崩溃中恢复
- 使用阈值（HWM）防止内存溢出

零的哲学

ØMQ一词中的Ø让我们纠结了很久。一方面，这个特殊字符会降低ZMQ在谷歌和推特中的收入量；另一方面，这会惹恼某些丹麦语种的民族，他们会嚷道Ø并不是一个奇怪的0。

一开始ZMQ代表零中间件、零延迟，同时，它又有了新的含义：零管理、零成本、零浪费。总的来说，零表示最小、最简，这是贯穿于该项目的哲理。我们致力于减少复杂程度，提高易用性。

套接字API

说实话，ZMQ有些偷梁换柱的嫌疑。不过我们并不会为此道歉，因为这种概念上的切换绝对不会有坏处。ZMQ提供了一套类似于BSD套接字的API，但将很多消息处理机制的细节隐藏了起来，你会逐渐适应这种变化，并乐于用它进行编程。

套接字事实上是用于网络编程的标准接口，ZMQ之所那么吸引人眼球，原因之一就是它是建立在标准套接字API之上。因此，ZMQ的套接字操作非常容易理解，其生命周期主要包含四个部分：

- 创建和销毁套接字：zmq_socket(), zmq_close()
- 配置和读取套接字选项：zmq_setsockopt(), zmq_getsockopt()
- 为套接字建立连接：zmq_bind(), zmq_connect()
- 发送和接收消息：zmq_send(), zmq_recv()

如以下C代码：

```
void *mousetrap;

//  Create socket for catching mice
mousetrap = zmq_socket (context, ZMQ_PULL);

//  Configure the socket
int64_t jawsize = 10000;
zmq_setsockopt (mousetrap, ZMQ_HWM, &jawsize, sizeof jawsize);

//  Plug socket into mouse hole
zmq_connect (mousetrap, "tcp://192.168.55.221:5001");

//  Wait for juicy mouse to arrive
zmq_msg_t mouse;
zmq_msg_init (&mouse);
zmq_recv (mousetrap, &mouse, 0);
//  Destroy the mouse
zmq_msg_close (&mouse);

//  Destroy the socket
zmq_close (mousetrap);
```

请注意，套接字永远是空指针类型的，而消息则是一个数据结构（我们下文会讲述）。所以，在C语言中你通过变量传递套接字，而用引用传递消息。记住一点，在ZMQ中所有的套接字都是由ZMQ管理的，只有消息是由程序员管理的。

创建、销毁、以及配置套接字的工作和处理一个对象差不多，但请记住ZMQ是异步的，伸缩性很强，因此在将其应用到网络结构中时，可能会需要多一些时间来理解。

使用套接字构建拓扑结构

在连接两个节点时，其中一个需要使用zmq_bind()，另一个则使用zmq_connect()。通常来讲，使用zmq_bind()连接的节点称之为服务端，它有着一个较为固定的网络地址；使用zmq_connect()连接的节点称为客户端，其地址不固定。我们会有这样的说法：绑定套接字至端点；连接套接字至端点。端点指的是某个广为周知网络地址。

ZMQ连接和传统的TCP连接是有区别的，主要有：

- 使用多种协议，inproc（进程内）、ipc（进程间）、tcp、pgm（广播）、epgm；

- 当客户端使用zmq_connect()时连接就已经建立了，并不要求该端点已有某个服务使用zmq_bind()进行了绑定；
- 连接是异步的，并由一组消息队列做缓冲；
- 连接会表现出某种消息模式，这是由创建连接的套接字类型决定的；
- 一个套接字可以有多个输入和输出连接；
- ZMQ没有提供类似zmq_accept()的函数，因为当套接字绑定至端点时它就自动开始接受连接了；
- 应用程序无法直接和这些连接打交道，因为它们是被封装在ZMQ底层的。

在很多架构中都使用了类似于C/S的架构。服务端组件式比较稳定的，而客户端组件则较为动态，来去自如。所以说，服务端地址对客户端而言往往是可见的，反之则不然。这样一来，架构中应该将哪些组件作为服务端（使用zmq_bind()），哪些作为客户端（使用zmq_connect()），就很明显了。同时，这需要和你使用的套接字类型相联系起来，我们下文会详细讲述。

让我们试想一下，如果先打开了客户端，后打开服务端，会发生什么？传统网络连接中，我们打开客户端时一定会收到系统的报错信息，但ZMQ让我们能够自由地启动架构中的组件。当客户端使用zmq_connect()连接至某个端点时，它就已经能够使用该套接字发送消息了。如果这时，服务端启动起来了，并使用zmq_bind()绑定至该端点，ZMQ将自动开始转发消息。

服务端节点可以仅使用一个套接字就能绑定至多个端点。也就是说，它能够使用不同的协议来建立连接：

```
zmq_bind (socket, "tcp://*:5555");
zmq_bind (socket, "tcp://*:9999");
zmq_bind (socket, "ipc://myserver.ipc");
```

当然，你不能多次绑定至同一端点，这样是会报错的。

每当有客户端节点使用zmq_connect()连接至上述某个端点时，服务端就会自动创建连接。ZMQ没有对连接数量进行限制。此外，客户端节点也可以使用一个套接字同时建立多个连接。

大多数情况下，哪个节点充当服务端，哪个作为客户端，是网络架构层面的内容，而非消息流问题。不过也有一些特殊情况（如失去连接后的消息重发），同一种套接字使用绑定和连接是会有一些不同的行为的。

所以说，当我们在设计架构时，应该遵循"服务端是稳定的，客户端是灵活的"原则，这样就不太会出错。

套接字是有类型的，套接字类型定义了套接字的行为，它在发送和接收消息时的规则等。你可以将不同种类的套接字进行连接，如PUB-SUB组合，这种组合称之为发布-订阅模式，其他组合也会有相应的模式名称，我们会在下文详述。

正是因为套接字可以使用不同的方式进行连接，才构成了ZMQ最基本的消息队列系统。我们还可以在此基础之上建立更为复杂的装置、路由机制等，下文会详述。总的来说，ZMQ为你提供了一套组件，供你在网络架构中拼装和使用。

使用套接字传递数据

发送和接收消息使用的是zmq_send()和zmq_recv()这两个函数。虽然函数名称看起来很直白，但由于ZMQ的I/O模式和传统的TCP协议有很大不同，因此还是需要花点时间去理解的。

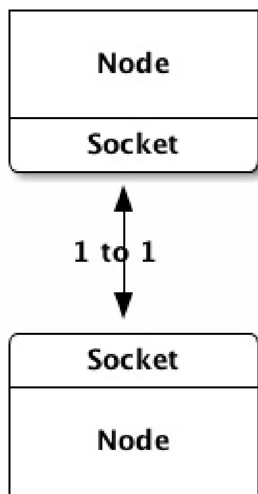


Figure 1 — TCP sockets are 1 to 1

让我们看一看TCP套接字和ZMQ套接字之间在传输数据方面的区别：

- ZMQ套接字传输的是消息，而不是字节（TCP）或帧（UDP）。消息指的是一段指定长度的二进制数据块，我们下文会讲到消息，这种设计是为了性能优化而考虑的，所以可能会比较难以理解。
- ZMQ套接字在后台进行I/O操作，也就是说无论是接收还是发送消息，它都会先传送到一个本地的缓冲队列，这个内存队列的大小是可以配置的。
- ZMQ套接字可以和多个套接字进行连接（如果套接字类型允许的话）。TCP协议只能进行点对点的连接，而ZMQ则可以进行一对多（类似于无线广播）、多对多（类似于邮局）、多对一（类似于信箱），当然也包括一对一的情况。
- ZMQ套接字可以发送消息给多个端点（扇出模型），或从多个端点中接收消息（扇入模型）

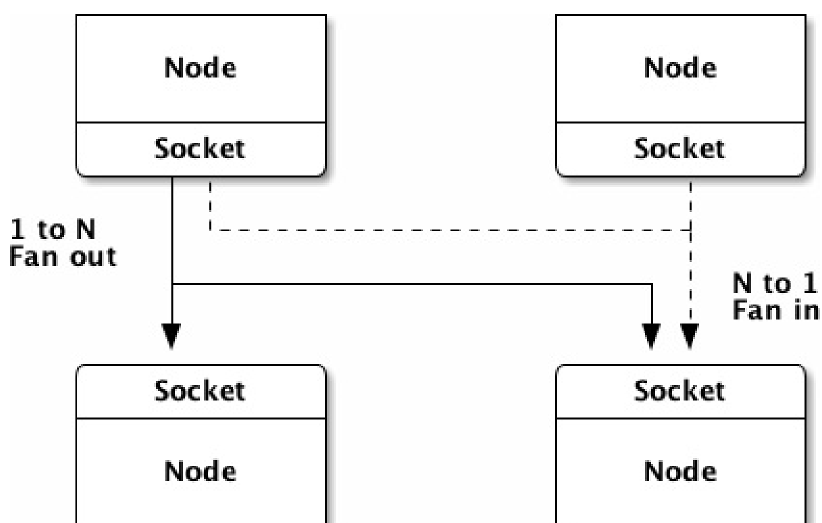


Figure 2 — 0MQ sockets are N to N

所以，向套接字写入一个消息时可能会将消息发送给很多节点，相应的，套接字又会从所有已建立的连接中接收消息。zmq_recv()方法使用了公平队列的算法来决定接收哪个连接的消息。

调用zmq_send()方法时其实并没有真正将消息发送给套接字连接。消息会在一个内存队列中保存下来，并由后台的I/O线程异步地进行发送。如果不出意外情况，这一行为是非阻塞的。所以说，即便zmq_send()有返回值，并不能代表消息已经发送。当你在用zmq_msg_init_data()初始化消息后，你不能重用或是释放这条消息，否则ZMQ的I/O线程会认为它在传输垃圾数据。这对初学者来讲是一个常犯的错误，下文我们会讲述如何正确地处理消息。

单播传输

ZMQ提供了一组单播传输协议（inporc, ipc, tcp），和两个广播协议（epgm, pgm）。广播协议是比较高级的协议，我们会在以后讲述。如果你不能回答我扇出比例会影响一对多的单播传输时，就先不要去学习广播协议了吧。

一般而言我们会使用**tcp**作为传输协议，这种TCP连接是可以脱机运作的，它灵活、便携、且足够快速。为什么称之为脱机，是因为ZMQ中的TCP连接不需要该端点已经有某个服务进行了绑定，客户端和服务端可以随时进行连接和绑定，这对应用程序而言都是透明的。

进程间协议，即**ipc**，和tcp的行为差不多，但已从网络传输中抽象出来，不需要指定IP地址或者域名。这种协议很多时候会很方便，本指南中的很多示例都会使用这种协议。ZMQ中的ipc协议同样可以是脱机的，但有一个缺点——无法在Windows操作系统上运作，这一点也许会在未来的ZMQ版本中修复。我们一般会在端点名称的末尾附上.ipc的扩展名，在UNIX系统上，使用ipc协议还需要注意权限问题。你还需要保证所有的程序都能够找到这个ipc端点。

进程内协议，即**inproc**，可以在同一个进程的不同线程之间进行消息传输，它比ipc或tcp要快得多。这种协议有一个要求，必须先绑定到端点，才能建立连接，也许未来也会修复。通常的做法是先启动服务端线程，绑定至端点，后启动客户端线程，连接至端点。

ZMQ不只是数据传输

经常有新人会问，如何使用ZMQ建立一项服务？我能使用ZMQ建立一个HTTP服务器吗？

他们期望得到的回答是，我们用普通的套接字来传输HTTP请求和应答，那用ZMQ套接字也能够完成这个任务，且能运行得更快、更好。

只可惜答案并不是这样的。ZMQ不只是一个数据传输的工具，而是在现有通信协议之上建立起来的新架构。它的数据帧和现有的协议并不兼容，如下面是一个HTTP请求和ZMQ请求的对比，同样使用的是TCP/IPC协议：

GET /index.html	13	10	13	10

Figure 3 — HTTP request

HTTP请求使用CR-LF（换行符）作为信息帧的间隔，而ZMQ则使用指定长度来定义帧：

5	H	E	L	L	O

Figure 4 — 0MQ request

所以说，你的确是可以用ZMQ来写一个类似于HTTP协议的东西，但是这并不是HTTP。

不过，如果有人问我如何更好地使用ZMQ建立一个新的服务，我会给出一个不错的答案，那就是：你可以自行设计一种通信协议，用ZMQ进行连接，使用不同的语言提供服务和扩展，可以在本地，亦可通过远程传输。赛德•肖的Mongrel2网络服务的架构就是一个很好的示例。

I/O线程

我们提过ZMQ是通过后台的I/O线程进行消息传输的。一个I/O线程已经足以处理多个套接字的数据传输要求，当然，那些极端的应用程序除外。这也就是我们在创建上下文时传入的1所代表的意思：

```
void *context = zmq_init (1);
```

ZMQ应用程序和传统应用程序的区别之一就是你不需要为每个套接字都创建一个连接。单个ZMQ套接字可以处理所有的发送和接收任务。如，当你需要向一千个订阅者发布消息时，使用一个套接字就可以了；当你需要向二十个服务进程分发任务时，使用一个套接字就可以了；当你需要从一千个网页应用程序中获取数据时，也是使用一个套接字就可以了。

这一特性可能会颠覆网络应用程序的编写步骤，传统应用程序每个进程或线程会有一个远程连接，它又只能处理一个套接字。ZMQ让你打破这种结构，使用一个线程来完成所有工作，更易于扩展。

核心消息模式

ZMQ的套接字API中提供了多种消息模式。如果你熟悉企业级消息应用，那这些模式会看起来很熟悉。不过对于新手来说，ZMQ的套接字还是会让人大吃一惊的。

让我们回顾一下ZMQ会为你做些什么：它会将消息快速高效地发送给其他节点，这里的节点可以是线程、进程、或是其他计算机；ZMQ为应用程序提供了一套简单的套接字API，不用考虑实际使用的协议类型（进程内、进程间、TPC、或广播）；当节点调动时，ZMQ会自动进行连接或重连；无论是发送消息还是接收消息，ZMQ都会先将消息放入队列中，并保证进程不会因为内存溢出而崩溃，适时地将消息写入磁盘；ZMQ会处理套接字异常；所有的I/O操作都在后台进行；ZMQ不会产生死锁。

但是，以上种种的前提是用户能够正确地使用消息模式，这种模式往往也体现出了ZMQ的智慧。消息模式将我们从实践中获取的经验进行抽象和重组，用于解决之后遇到的所有问题。ZMQ的消息模式目前是编译在类库中的，不过未来的ZMQ版本可能会允许用户自行制定消息模式。

ZMQ的消息模式是指不同类型套接字的组合。换句话说，要理解ZMQ的消息模式，你需要理解ZMQ的套接字类型，它们是如何一起工作的。这一部分是需要死记硬背的。

ZMQ的核心消息模式有：

- 请求-应答模式 将一组服务端和一组客户端相连，用于远程过程调用或任务分发。
- 发布-订阅模式 将一组发布者和一组订阅者相连，用于数据分发。
- 管道模式 使用扇入或扇出的形式组装多个节点，可以产生多个步骤或循环，用于构建并行处理架构。

我们在第一章中已经讲述了这些模式，不过还有一种模式是为那些仍然认为ZMQ是类似TCP那样点对点连接的人们准备的：

- 排他对接模式 将两个套接字一对一地连接起来，这种模式应用场景很少，我们会在本章最末尾看到一个示例。

zmq_socket()函数的说明页中有对所有消息模式的说明，比较清楚，因此值得研读几次。我们会介绍每种消息模式的内容和应用场景。

以下是合法的套接字连接-绑定对（一端绑定、一端连接即可）：

- PUB - SUB
- REQ - REP
- REQ - ROUTER
- DEALER - REP
- DEALER - ROUTER
- DEALER - DEALER
- ROUTER - ROUTER
- PUSH - PULL
- PAIR - PAIR

其他的组合模式会产生不可预知的结果，在将来的ZMQ版本中可能会直接返回错误。你也可以通过代码去了解这些套接字类型的行为。

上层消息模式

上文中的四种核心消息模式是内建在ZMQ中的，他们是API的一部分，在ZMQ的C++核心类库中实现，能够保证正确地运行。如果有朝一日Linux内核将ZMQ采纳了进来，那这些核心模式也肯定会包含其中。

在这些消息模式之上，我们会建立更为上层的消息模式。这种模式可以用任何语言编写，他们不属于核心类型的一部分，不随ZMQ发行，只在你自己的应用程序中出现，或者在ZMQ社区中维护。

本指南的目的之一就是为你提供一些上层的消息模式，有简单的（如何正确处理消息），也有复杂的（可靠的发布-订阅模式）。

消息的使用方法

ZMQ的传输单位是消息，即一个二进制块。你可以使用任意的序列化工具，如谷歌的Protocal Buffers、XDR、JSON等，将内容转化成ZMQ消息。不过这种转化工具最好是便捷和快速的，这个请自己衡量。

在内存中，ZMQ消息由zmq_msg_t结构表示（每种语言有特定的表示）。在C语言中使用ZMQ消息时需要注意以下几点：

- 你需要创建和传递zmq_msg_t对象，而不是一组数据块；
- 读取消息时，先用zmq_msg_init()初始化一个空消息，再将其传递给zmq_recv()函数；
- 写入消息时，先用zmq_msg_init_size()来创建消息（同时也已初始化了一块内存区域），然后用memcpy()函数将信息拷贝到该对象中，最后传给zmq_send()函数；
- 释放消息（并不是销毁）时，使用zmq_msg_close()函数，它会将对消息对象的引用删除，最终由ZMQ将消息销毁；
- 获取消息内容时需使用zmq_msg_data()函数；若想知道消息的长度，可以使用zmq_msg_size()函数；
- 至于zmq_msg_move()、zmq_msg_copy()、zmq_msg_init_data()函数，在充分理解手册中的说明之前，建议不好贸然使用。

以下是一段处理消息的典型代码，如果之前的代码你有看的话，那应该会感到熟悉。这段代码其实是从zhelpers.h文件中抽出的：

```
//  从套接字中获取ZMQ字符串，并转换为C语言字符串
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_recv (socket, &message, 0);
    int size = zmq_msg_size (&message);
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}

//  将C语言字符串转换为ZMQ字符串，并发送给套接字
static int
s_send (void *socket, char *string) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    rc = zmq_send (socket, &message, 0);
    assert (!rc);
    zmq_msg_close (&message);
    return (rc);
}
```

你可以对以上代码进行扩展，让其支持发送和接受任一长度的数据。

需要注意的是，当你将一个消息对象传递给**zmq_send()**函数后，该对象的长度就会被清零，因此你无法发送同一个消息对象两次，也无法获得已发送消息的内容。

如果你想发送同一个消息对象两次，就需要在发送第一次前新建一个对象，使用zmq_msg_copy()函数进行拷贝。这个函数不会拷贝消息内容，只是拷贝引用。然后你就可以再次发送这个消息了（或者任意多次，只要进行了足够的拷贝）。当消息最后一个引用被释放时，消息对象就会被销毁。

ZMQ支持多帧消息，即在一条消息中保存多个消息帧。这在实际应用中被广泛使用，我们会在第三章进行讲解。

关于消息，还有一些需要注意的地方：

- ZMQ的消息是作为一个整体来收发的，你不会只收到消息的一部分；
- ZMQ不会立即发送消息，而是有一定的延迟；
- 你可以发送0字节长度的消息，作为一种信号；
- 消息必须能够在内存中保存，如果你想发送文件或超长的消息，就需要将他们切割成小块，在独立的消息中进行发送；
- 必须使用zmq_msg_close()函数来关闭消息，但在一些会在变量超出作用域时自动释放消息对象的语言中除外。

再重复一句，不要贸然使用zmq_msg_init_data()函数。它是用于零拷贝，而且可能会造成麻烦。关于ZMQ还有太多东西需要你去学习，因此现在暂时不用去考虑如何削减几微秒的开销。

处理多个套接字

在之前的示例中，主程序的循环体内会做以下几件事：

1. 等待套接字的消息；
2. 处理消息；
3. 返回第一步。

如果我们想要读取多个套接字中的消息呢？最简单的方法是将套接字连接到多个端点上，让ZMQ使用公平队列的机制来接受消息。如果不同端点上的套接字类型是一致的，那可以使用这种方法。但是，如果一个套接字的类型是PULL，另一个是PUB怎么办？如果现在开始混用套接字类型，那将来就没有可靠性可言了。

正确的方法应该是使用zmq_poll()函数。更好的方法是将zmq_poll()包装成一个框架，编写一个事件驱动的反应器，但这个就比较复杂了，我们这里暂不讨论。

我们先不使用zmq_poll()，而用NOBLOCK（非阻塞）的方式来实现从多个套接字读取消息的功能。下面将气象信息服务和并行处理这两个示例结合起来：

msreader: Multiple socket reader in C

```
//
//  从多个套接字中获取消息
//  本示例简单地再循环中使用recv函数
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);

    //  连接至任务分发器
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  连接至天气服务
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    //  处理从两个套接字中接收到的消息
    //  这里我们会优先处理从任务分发器接收到的消息
    while (1) {
        //  处理等待中的任务
        int rc;
        for (rc = 0; !rc; ) {
```

```
                zmq_msg_t task;
                zmq_msg_init (&task);
                if ((rc = zmq_recv (receiver, &task, ZMQ_NOBLOCK)) == 0
                    //  处理任务
                }
                zmq_msg_close (&task);
            }
            //  处理等待中的气象更新
            for (rc = 0; !rc; ) {
                zmq_msg_t update;
                zmq_msg_init (&update);
                if ((rc = zmq_recv (subscriber, &update, ZMQ_NOBLOCK))
                    //  处理气象更新
                }
                zmq_msg_close (&update);
            }
            // 没有消息，等待1毫秒
            s_sleep (1);
        }
        //  程序不会运行到这里，但还是做正确的退出清理工作
        zmq_close (receiver);
        zmq_close (subscriber);
        zmq_term (context);
        return 0;
    }
```

这种方式的缺点之一是，在收到第一条消息之前会有1毫秒的延迟，这在高压力的程序中还是会构成问题的。此外，你还需要翻阅诸如nanosleep()的函数，不会造成循环次数的激增。

示例中将任务分发器的优先级提升了，你可以做一个改进，轮流处理消息，正如ZMQ内部做的公平队列机制一样。

下面，让我们看看如何用zmq_poll()来实现同样的功能：

mspoller: Multiple socket poller in C

```c
//
//  从多个套接字中接收消息
//  本例使用zmq_poll()函数
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  连接任务分发器
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  连接气象更新服务
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    //  初始化轮询对象
    zmq_pollitem_t items [] = {
        { receiver, 0, ZMQ_POLLIN, 0 },
        { subscriber, 0, ZMQ_POLLIN, 0 }
    };
    //  处理来自两个套接字的消息
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_recv (receiver, &message, 0);
            //  处理任务
            zmq_msg_close (&message);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_recv (subscriber, &message, 0);
            //  处理气象更新
            zmq_msg_close (&message);
        }
    }
    //  程序不会运行到这儿
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

处理错误和ETERM信号

ZMQ的错误处理机制提倡的是快速崩溃。我们认为，一个进程对于自身内部的错误来说要越脆弱越好，而对外部的攻击和错误要足够健壮。举个例子，活细胞会因检测到自身问题而瓦解，但对外界的攻击却能极力抵抗。在ZMQ编程中，断言用得是非常多的，如同细胞膜一样。如果我们无法确定一个错误是来自于内部还是外部，那这就是一个设计缺陷了，需要修复。

在C语言中，断言失败会让程序立即中止。其他语言中可以使用异常来做到。

当ZMQ检测到来自外部的问题时，它会返回一个错误给调用程序。如果ZMQ不能从错误中恢复，那它是不会安静地将消息丢弃的。某些情况下，ZMQ也会去断言外部错误，这些可以被归结为BUG。

到目前为止，我们很少看到C语言的示例中有对错误进行处理。现实中的代码应该对每一次的**ZMQ**函数调用作错误处理。如果你不是使用C语言进行编程，可能那种语言的ZMQ类库已经做了错误处理。但在C语言中，你需要自己动手。以下是一些常规的错误处理手段，从POSIX规范开始：

- 创建对象的方法如果失败了会返回NULL；
- 其他方法执行成功时会返回0，失败时会返回其他值（一般是-1）；
- 错误代码可以从变量errno中获得，或者调用zmq_errno()函数；
- 错误消息可以调用zmq_strerror()函数获得。

有两种情况不应该被认为是错误：

- 当线程使用NOBLOCK方式调用zmq_recv()时，若没有接收到消息，该方法会返回-1，并设置errno为EAGAIN；
- 当线程调用zmq_term()时，若其他线程正在进行阻塞式的处理，该函数会中止所有的处理，关闭套接字，并使得那些阻塞方法的返回值为-1，errno设置为ETERM。

遵循以上规则，你就可以在ZMQ程序中使用断言了：

```
void *context = zmq_init (1);
assert (context);
void *socket = zmq_socket (context, ZMQ_REP);
assert (socket);
int rc;
rc = zmq_bind (socket, "tcp://*:5555");
assert (rc == 0);
```

第一版的程序中我将函数调用直接放在了assert()函数里面，这样做会有问题，因为一些优化程序会直接将程序中的assert()函数去除。

让我们看看如何正确地关闭一个进程，我们用管道模式举例。当我们在后台开启了一组worker时，我们需要在任务执行完毕后关闭它们。我们可以向这些worker发送自杀的消息，这项工作由结果收集器来完成会比较恰当。

如何将结果收集器和worker相连呢？PUSH-PULL套接字是单向的。ZMQ的原则是：如果需要解决一个新的问题，就该使用新的套接字。这里我们使用发布-订阅模式来发送自杀的消息：

- 结果收集器创建PUB套接字，并连接至一个新的端点；
- worker将SUB套接字连接至这个端点；
- 当结果收集器检测到任务执行完毕时，会通过PUB套接字发送自杀信号；
- worker收到自杀信号后便会中止。

这一过程不会添加太多的代码：

```
void *control = zmq_socket (context, ZMQ_PUB);
zmq_bind (control, "tcp://*:5559");
...
//  Send kill signal to workers
zmq_msg_init_data (&message, "KILL", 5);
zmq_send (control, &message, 0);
zmq_msg_close (&message);
```

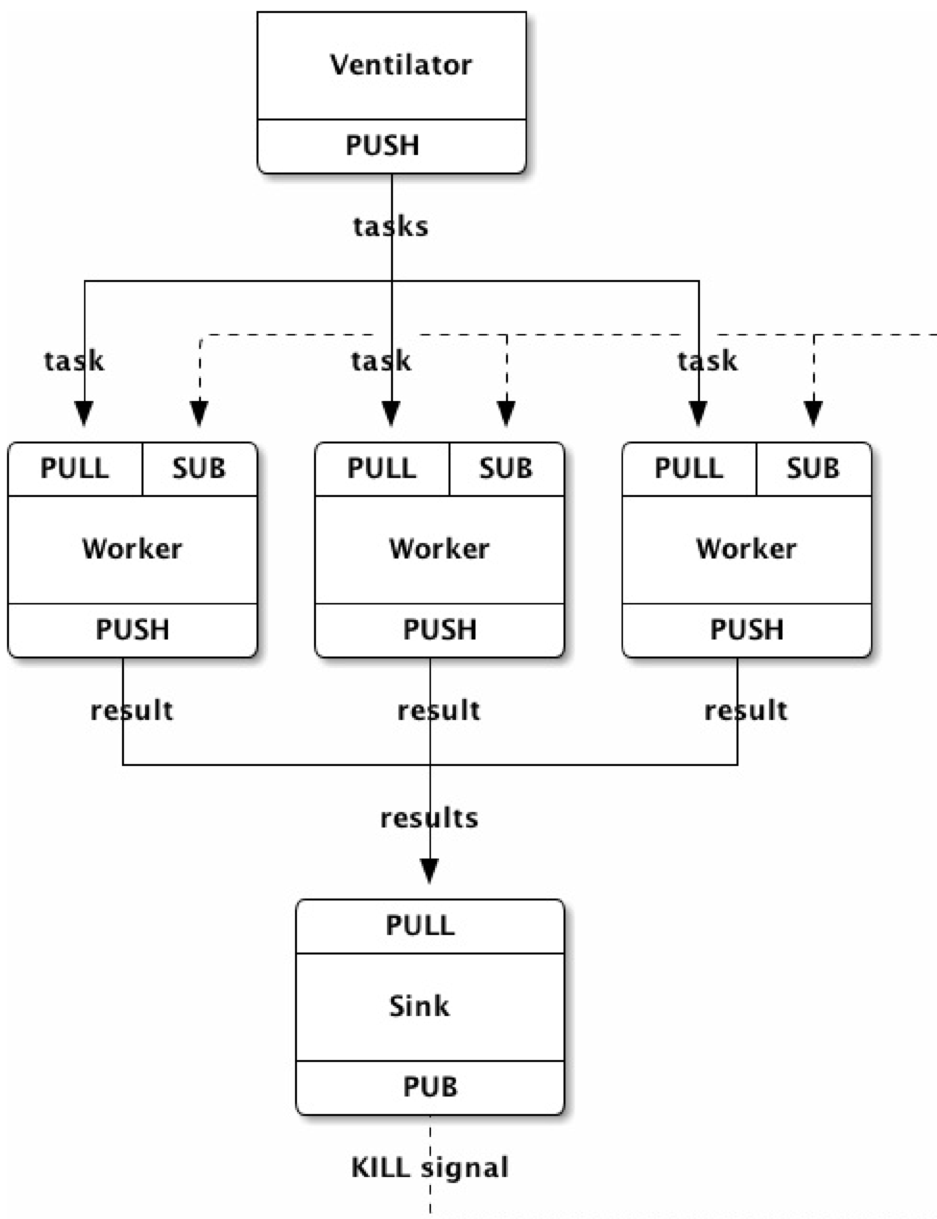


Figure 5 — Parallel Pipeline with Kill signaling

下面是worker进程的代码，它会打开三个套接字：用于接收任务的PULL、用于发送结果的PUSH、以及用于接收自杀信号的SUB，使用zmq_poll()进行轮询：

taskwork2: Parallel task worker with kill signaling in C

```
//
//  管道模式 - worker 设计2
//  添加发布-订阅消息流，用以接收自杀消息
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  用户发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  用户接收控制消息的套接字
    void *controller = zmq_socket (context, ZMQ_SUB);
    zmq_connect (controller, "tcp://localhost:5559");
    zmq_setsockopt (controller, ZMQ_SUBSCRIBE, "", 0);

    //  处理接收到的任务或控制消息
    zmq_pollitem_t items [] = {
        { receiver, 0, ZMQ_POLLIN, 0 },
        { controller, 0, ZMQ_POLLIN, 0 }
    };
    //  处理消息
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_recv (receiver, &message, 0);

            //  工作
            s_sleep (atoi ((char *) zmq_msg_data (&message)));

            //  发送结果
            zmq_msg_init (&message);
            zmq_send (sender, &message, 0);

            //  简单的任务进图指示
            printf (".");
            fflush (stdout);
```

```
                zmq_msg_close (&message);
            }
            //  任何控制命令都表示自杀
            if (items [1].revents & ZMQ_POLLIN)
                break;                      //  退出循环
        }
        //  结束程序
        zmq_close (receiver);
        zmq_close (sender);
        zmq_close (controller);
        zmq_term (context);
        return 0;
    }
```

下面是修改后的结果收集器代码，在收集完结果后向所有worker发送自杀消息：

tasksink2: Parallel task sink with kill signaling in C

```
//
//  管道模式 - 结构收集器 设计2
//  添加发布-订阅消息流，用以向worker发送自杀信号
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    //  用以发送控制信息的套接字
    void *controller = zmq_socket (context, ZMQ_PUB);
    zmq_bind (controller, "tcp://*:5559");

    //  等待任务开始
    char *string = s_recv (receiver);
    free (string);

    //  开始计时
    int64_t start_time = s_clock ();

    //  确认100个任务处理完毕
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    printf ("总执行时间: %d msec\n",
        (int) (s_clock () - start_time));

    //  发送自杀消息给worker
    s_send (controller, "KILL");

    //  结束
    sleep (1);              //  等待发送完毕

    zmq_close (receiver);
    zmq_close (controller);
    zmq_term (context);
    return 0;
}
```

处理中断信号

现实环境中，当应用程序收到Ctrl-C或其他诸如ETERM的信号时需要能够正确地清理和退出。默认情况下，这一信号会杀掉进程，意味着尚未发送的消息就此丢失，文件不能被正确地关闭等。

在C语言中我们是这样处理消息的：

interrupt: Handling Ctrl-C cleanly in C

```
//
//  Shows how to handle Ctrl-C
//
#include <zmq.h>
#include <stdio.h>
#include <signal.h>

//  ---------------------------------------------------------------------
//  消息处理
//
//  程序开始运行时调用s_catch_signals()函数；
//  在循环中判断s_interrupted是否为1，是则跳出循环；
//  很适用于zmq_poll()。

static int s_interrupted = 0;
static void s_signal_handler (int signal_value)
{
    s_interrupted = 1;
}

static void s_catch_signals (void)
{
    struct sigaction action;
    action.sa_handler = s_signal_handler;
    action.sa_flags = 0;
    sigemptyset (&action.sa_mask);
    sigaction (SIGINT, &action, NULL);
    sigaction (SIGTERM, &action, NULL);
}

int main (void)
{
    void *context = zmq_init (1);
    void *socket = zmq_socket (context, ZMQ_REP);
    zmq_bind (socket, "tcp://*:5555");

    s_catch_signals ();
    while (1) {
        //  阻塞式的读取会在收到信号时停止
        zmq_msg_t message;
        zmq_msg_init (&message);
        zmq_recv (socket, &message, 0);
```

```
        if (s_interrupted) {
            printf ("W: 收到中断消息，程序中止...\n");
            break;
        }
    }
    zmq_close (socket);
    zmq_term (context);
    return 0;
}
```

这段程序使用s_catch_signals()函数来捕捉像Ctrl-C(SIGINT)和SIGTERM这样的信号。收到任一信号后，该函数会将全局变量s_interrupted设置为1。你的程序并不会自动停止，需要显式地做一些清理和退出工作。

- 在程序开始时调用s_catch_signals()函数，用来进行信号捕捉的设置；
- 如果程序在zmq_recv()、zmq_poll()、zmq_send()等函数中阻塞，当有信号传来时，这些函数会返回EINTR；
- 像s_recv()这样的函数会将这种中断包装为NULL返回；
- 所以，你的应用程序可以检查是否有EINTR错误码、或是NULL的返回、或者s_interrupted变量是否为1。

如果以下代码就十分典型：

```
s_catch_signals ();
client = zmq_socket (...);
while (!s_interrupted) {
    char *message = s_recv (client);
    if (!message)
        break;          //  按下了Ctrl-C
}
zmq_close (client);
```

如果你在设置s_catch_signals()之后没有进行相应的处理，那么你的程序将对Ctrl-C和ETERM免疫。

检测内存泄露

任何长时间运行的程序都应该妥善的管理内存，否则最终会发生内存溢出，导致程序崩溃。如果你所使用的编程序言会自动帮你完成内存管理，那就要恭喜你了。但若你使用类似C/C++之类的语言时，就需要自己动手进行内存管理了。下面会介绍一个名为valgrind的工具，可以用它来报告内存泄露的问题。

- 在Ubuntu或Debian操作系统上安装valgrind：sudo apt-get install valgrind

- 缺省情况下，ZMQ会让valgrind不停地报错，想要屏蔽警告的话可以在编译ZMQ时使用ZMQ_MAKE_VALGRIND_HAPPY宏选项：

```
$ cd zeromq2
$ export CPPFLAGS=-DZMQ_MAKE_VALGRIND_HAPPY
$ ./configure
$ make clean; make
$ sudo make install
```

- 应用程序应该正确地处理Ctrl-C，特别是对于长时间运行的程序（如队列装置），如果不这么做，valgrind会报告所有已分配的内存发生了错误。
- 使用-DDEBUG选项编译程序，这样可以让valgrind告诉你具体是哪段代码发生了内存溢出。
- 最后，使用如下方法运行valgrind：

```
valgrind --tool=memcheck --leak-check=full someprog
```

解决完所有的问题后，你会看到以下信息：

```
==30536== ERROR SUMMARY: 0 errors from 0 contexts...
```

多帧消息

ZMQ消息可以包含多个帧，这在实际应用中非常常见，特别是那些有关“信封”的应用，我们下文会谈到。我们这一节要讲的是如何正确地收发多帧消息。

多帧消息的每一帧都是一个zmq_msg结构，也就是说，当你在收发含有五个帧的消息时，你需要处理五个zmq_msg结构。你可以将这些帧放入一个数据结构中，或者直接一个个地处理它们。

下面的代码演示如何发送多帧消息：

```
zmq_send (socket, &message, ZMQ_SNDMORE);
...
zmq_send (socket, &message, ZMQ_SNDMORE);
...
zmq_send (socket, &message, 0);
```

然后我们看看如何接收并处理这些消息，这段代码对单帧消息和多帧消息都适用：

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_recv (socket, &message, 0);
    // 处理一帧消息
    zmq_msg_close (&message);
    int64_t more;
    size_t more_size = sizeof (more);
    zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
    if (!more)
        break; // 已到达最后一帧
}
```

关于多帧消息，你需要了解的还有：

- 在发送多帧消息时，只有当最后一帧提交发送了，整个消息才会被发送；
- 如果使用了zmq_poll()函数，当收到了消息的第一帧时，其它帧其实也已经收到了；
- 多帧消息是整体传输的，不会只收到一部分；
- 多帧消息的每一帧都是一个zmq_msg结构；
- 无论你是否检查套接字的ZMQ_RCVMORE选项，你都会收到所有的消息；
- 发送时，ZMQ会将开始的消息帧缓存在内存中，直到收到最后一帧才会发送；
- 我们无法在发送了一部分消息后取消发送，只能关闭该套接字。

中间件和装置

当网络组件的数量较少时，所有节点都知道其它节点的存在。但随着节点数量的增加，这种结构的成本也会上升。因此，我们需要将这些组件拆分成更小的模块，使用一个中间件来连接它们。

这种结构在现实世界中是非常常见的，我们的社会和经济体系中充满了中间件的机制，用以降低复杂度，压缩构建大型网络的成本。中间件也会被称为批发商、分包商、管理者等等。

ZMQ网络也是一样，如果规模不断增长，就一定会需要中间件。ZMQ中，我们称其为“装置”。在构建ZMQ软件的初期，我们会画出几个节点，然后将它们连接起来，不使用中间件：

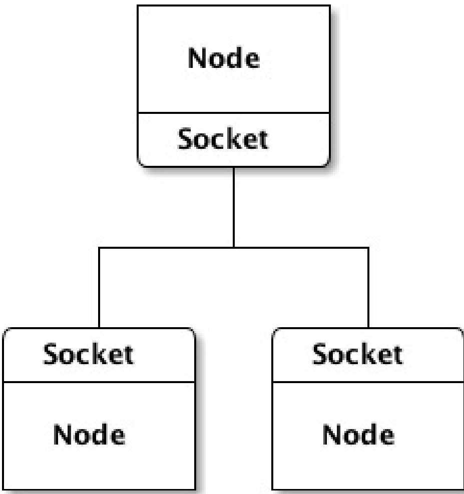


Figure 6 — Small scale 0MQ application

随后，我们对这个结构不断地进行扩充，将装置放到特定的位置，进一步增加节点数量：

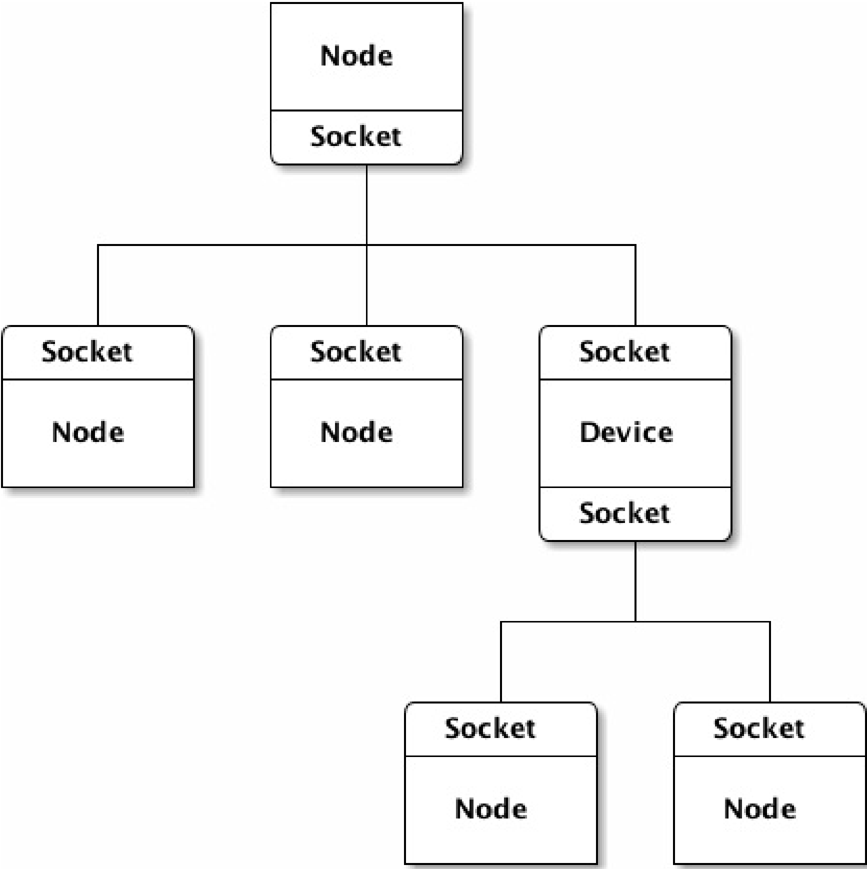


Figure 7 — Larger scale 0MQ application

ZMQ装置没有具体的设计规则，但一般会有一组“前端”端点和一组“后端”端点。装置是无状态的，因此可以被广泛地部署在网络中。你可以在进程中启动一个线程来运行装置，或者直接在一个进程中运行装置。ZMQ内部也提供了基本的装置实现可供使用。

ZMQ装置可以用作路由和寻址、提供服务、队列调度、以及其他你所能想到的事情。不同的消息模式需要用到不同类型的装置来构建网络。如，请求-应答模式中可以使用队列装置、抽象服务；发布-订阅模式中则可使用流装置、主题装置等。

ZMQ装置比起其他中间件的优势在于，你可以将它放在网络中任何一个地方，完成任何你想要的事情。

发布-订阅代理服务

我们经常会需要将发布-订阅模式扩充到不同类型的网络中。比如说，有一组订阅者是在外网上的，我们想用广播的方式发布消息给内网的订阅者，而用TCP协议发送给外网订阅者。

我们要做的就是写一个简单的代理服务装置，在发布者和外网订阅者之间搭起桥梁。这个装置有两个端点，一端连接内网上的发布者，另一端连接到外网上。它会从发布者处接收订阅的消息，并转发给外网上的订阅者们。

wuproxy: Weather update proxy in C

```
//
//  气象信息代理服务装置
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  订阅气象信息
    void *frontend = zmq_socket (context, ZMQ_SUB);
    zmq_connect (frontend, "tcp://192.168.55.210:5556");

    //  转发气象信息
    void *backend = zmq_socket (context, ZMQ_PUB);
    zmq_bind (backend, "tcp://10.1.1.0:8100");

    //  订阅所有消息
    zmq_setsockopt (frontend, ZMQ_SUBSCRIBE, "", 0);

    //  转发消息
    while (1) {
        while (1) {
            zmq_msg_t message;
            int64_t more;

            //  处理所有的消息帧
            zmq_msg_init (&message);
            zmq_recv (frontend, &message, 0);
            size_t more_size = sizeof (more);
            zmq_getsockopt (frontend, ZMQ_RCVMORE, &more, &more_siz
            zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  到达最后一帧
        }
    }
    //  程序不会运行到这里，但依然要正确地退出
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

我们称这个装置为代理，因为它既是订阅者，又是发布者。这就意味着，添加该装置时不需要更改其他程序的代码，只需让外网订阅者知道新的网络地址即可。

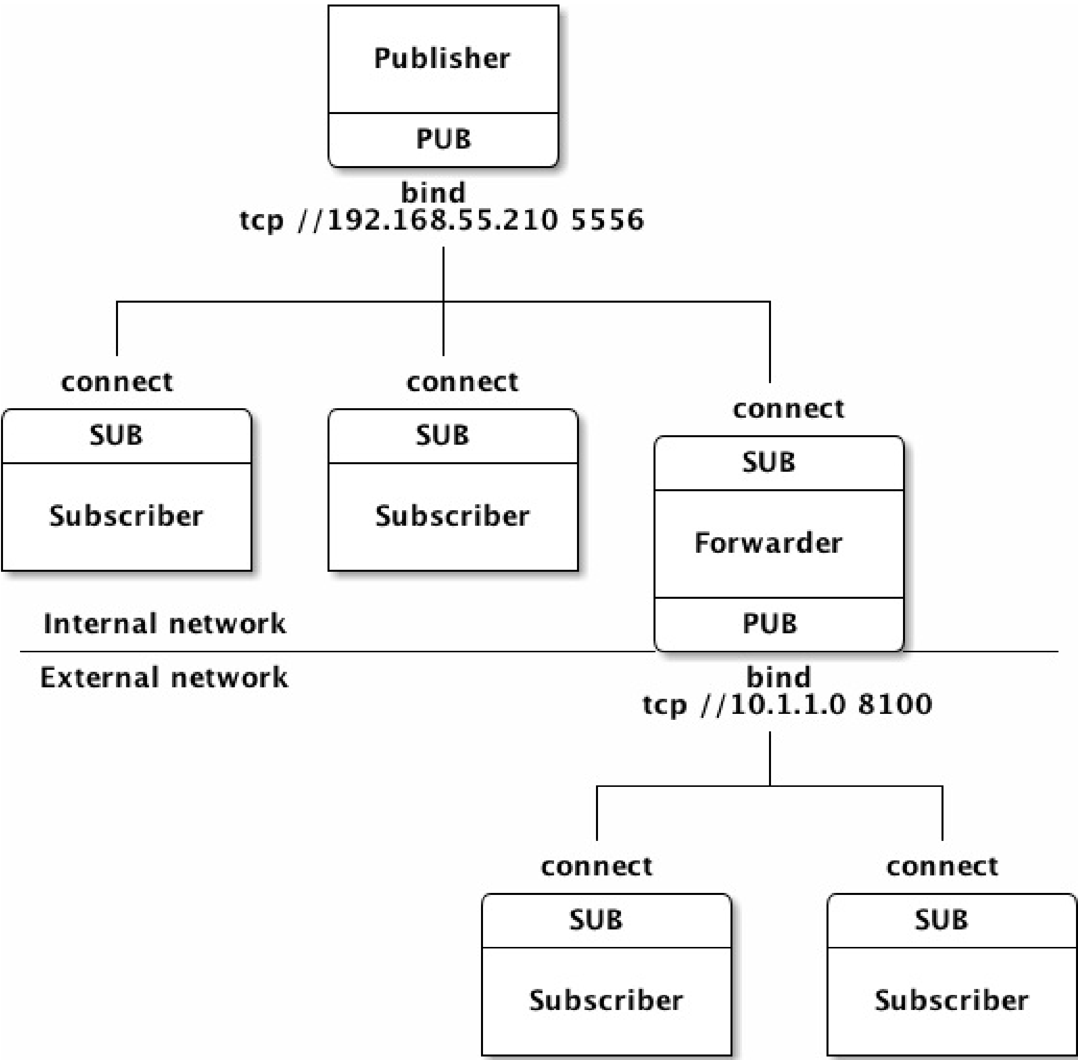


Figure 8 — Forwarder proxy device

可以注意到，这段程序能够正确处理多帧消息，会将它完整的转发给订阅者。如果我们在发送时不指定ZMQ_SNDMORE选项，那么下游节点收到的消息就可能是破损的。编写装置时应该要保证能够正确地处理多帧消息，否则会造成消息的丢失。

请求-应答代理

下面让我们在请求-应答模式中编写一个小型的消息队列代理装置。

在Hello World客户/服务模型中，一个客户端和一个服务端进行通信。但在真实环境中，我们会需要让多个客户端和多个服务端进行通信。关键问题在于，服务端应该是无状态的，所有的状态都应该包含在一次请求中，或者存放其它介质中，如数据库。

我们有两种方式来连接多个客户端和多个服务端。第一种是让客户端直接和多个服务端进行连接。客户端套接字可以连接至多个服务端套接字，它所发送的请求会通过负载均衡的方式分发给服务端。比如说，有一个客户端连接了三个服务端，A、B、C，客户端产生了R1、R2、R3、R4四个请求，那么，R1和R4会由服务A处理，R2由B处理，R3由C处理：

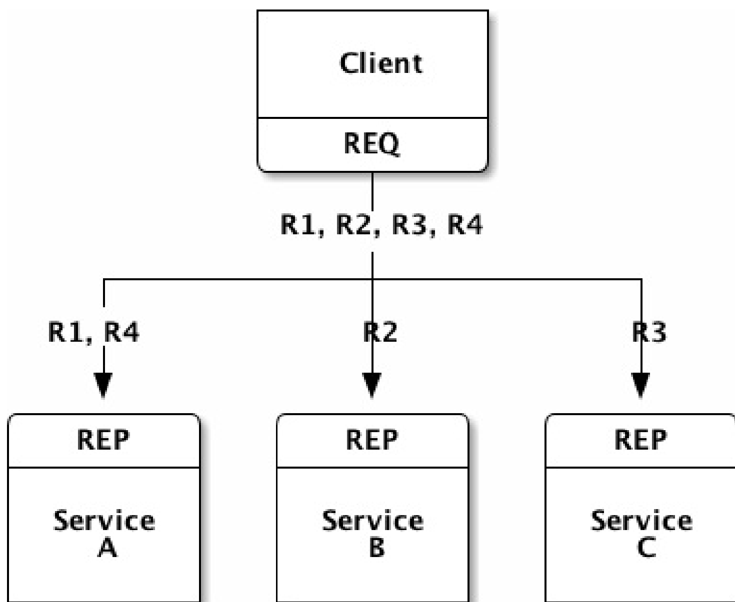


Figure 9 — Load balancing of requests

这种设计的好处在于可以方便地添加客户端，但若要添加服务端，那就得修改每个客户端的配置。如果你有100个客户端，需要添加三个服务端，那么这些客户端都需要重新进行配置，让其知道新服务端的存在。

这种方式肯定不是我们想要的。一个网络结构中如果有太多固化的模块就越不容易扩展。因此，我们需要有一个模块位于客户端和服务端之间，将所有的知识都汇聚到这个网络拓扑结构中。理想状态下，我们可以任意地增减客户端或是服务端，不需要更改任何组件的配置。

下面就让我们编写这样一个组件。这个代理会绑定到两个端点，前端端点供客户端连接，后端端点供服务端连接。它会使用zmq_poll()来轮询这两个套接字，接收消息并进行转发。装置中不会有队列的存在，因为ZMQ已经自动在套接字中完成了。

在使用REQ和REP套接字时，其请求-应答的会话是严格同步。客户端发送请求，服务端接收请求并发送应答，由客户端接收。如果客户端或服务端中的一个发生问题（如连续两次发送请求），程序就会报错。

但是，我们的代理装置必须要是非阻塞式的，虽然可以使用zmq_poll()同时处理两个套接字，但这里显然不能使用REP和REQ套接字。

幸运的是，我们有DEALER和ROUTER套接字可以胜任这项工作，进行非阻塞的消息收发。DEALER过去被称为XREQ，ROUTER被称为XREP，但新的代码中应尽量使用DEALER/ROUTER这种名称。在第三章中你会看到如何用DEALER和ROUTER套接字构建不同类型的请求-应答模式。

下面就让我们看看DEALER和ROUTER套接字是怎样在装置中工作的。

下方的简图描述了一个请求-应答模式，REQ和ROUTER通信，DEALER再和REP通信。ROUTER和DEALER之间我们则需要进行消息转发：

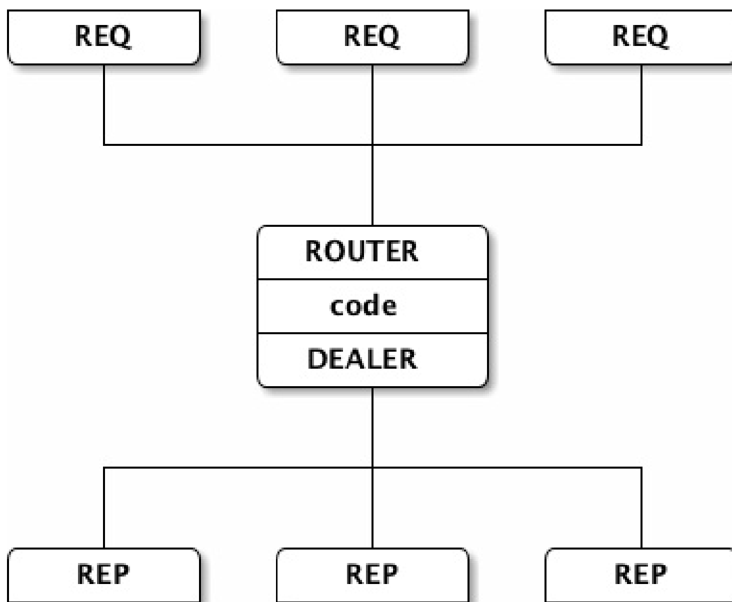


Figure 10 — Extended request reply

请求-应答代理会将两个套接字分别绑定到前端和后端，供客户端和服务端套接字连接。在使用该装置之前，还需要对客户端和服务端的代码进行调整。

rrclient: Request-reply client in C

```
//
//  Hello world 客户端
//  连接REQ套接字至 tcp://localhost:5559 端点
//  发送Hello给服务端，等待World应答
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  用于和服务端通信的套接字
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5559");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        s_send (requester, "Hello");
        char *string = s_recv (requester);
        printf ("收到应答 %d [%s]\n", request_nbr, string);
        free (string);
    }
    zmq_close (requester);
    zmq_term (context);
    return 0;
}
```

下面是服务代码：

rrserver: Request-reply service in C

```
//
//  Hello World 服务端
//  连接REP套接字至 tcp://*:5560 端点
//  接收Hello请求，返回World应答
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  用于何客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_connect (responder, "tcp://localhost:5560");

    while (1) {
        //  等待下一个请求
        char *string = s_recv (responder);
        printf ("Received request: [%s]\n", string);
        free (string);

        //  做一些“工作”
        sleep (1);

        //  返回应答信息
        s_send (responder, "World");
    }
    //  程序不会运行到这里，不过还是做好清理工作
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

最后是代理程序，可以看到它是能够处理多帧消息的：

rrbroker: Request-reply broker in C

```
//
//  简易请求-应答代理
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
```

```
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

    //  初始化轮询集合
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  在套接字间转发消息
    while (1) {
        zmq_msg_t message;
        int64_t more;           //  检测多帧消息

        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理所有消息帧
                zmq_msg_init (&message);
                zmq_recv (frontend, &message, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (frontend, ZMQ_RCVMORE, &more, &more
                zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  最后一帧
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理所有消息帧
                zmq_msg_init (&message);
                zmq_recv (backend, &message, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (backend, ZMQ_RCVMORE, &more, &more_
                zmq_send (frontend, &message, more? ZMQ_SNDMORE: 0)
                zmq_msg_close (&message);
                if (!more)
                    break;      //  最后一帧
            }
        }
    }
    //  程序不会运行到这里，不过还是做好清理工作
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

使用请求-应答代理可以让你的C/S网络结构更易于扩展：客户端不知道服务端的存在，服务端不知道客户端的存在。网络中唯一稳定的组件是中间的代理装置：

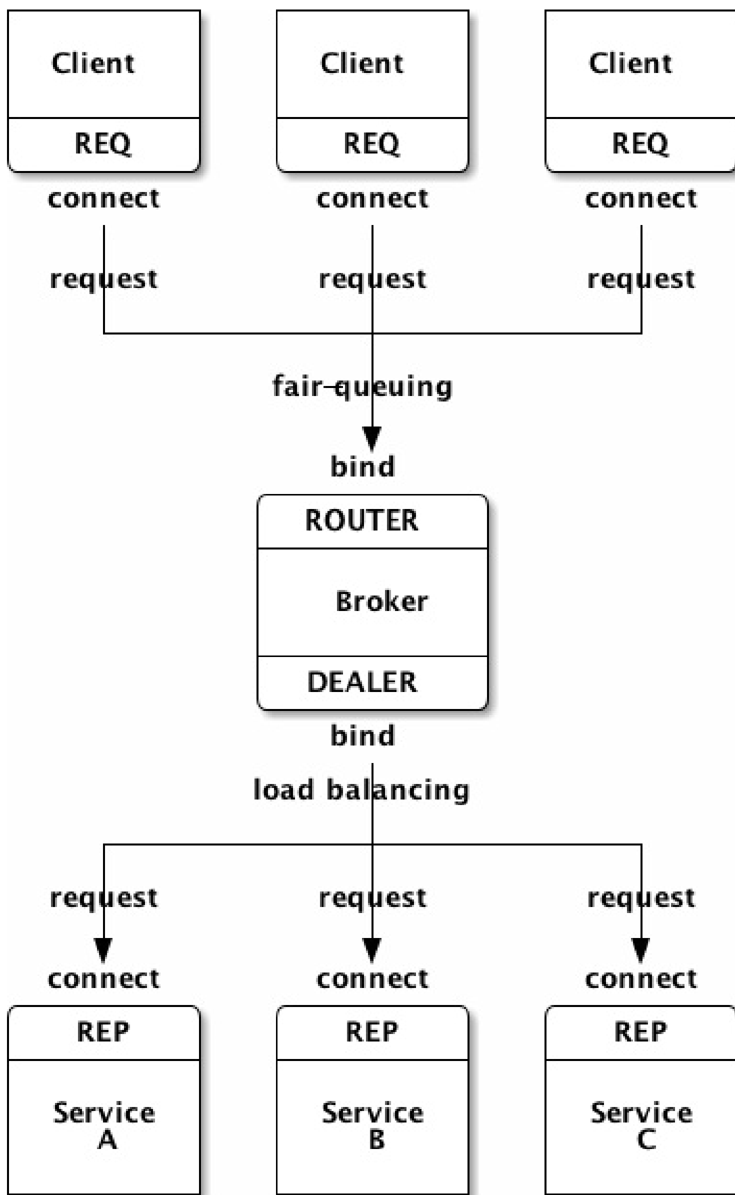


Figure 11 — Request reply broker

内置装置

ZMQ提供了一些内置的装置，不过大多数人需要自己手动编写这些装置。内置装置有：

- QUEUE，可用作请求-应答代理；
- FORWARDER，可用作发布-订阅代理服务；
- STREAMER，可用作管道模式代理。

可以使用zmq_device()来启动一个装置，需要传递两个套接字给它：

```
zmq_device (ZMQ_QUEUE, frontend, backend);
```

启动了QUEUE队列就如同在网络中加入了一个请求-应答代理，只需为其创建已绑定或连接的套接字即可。下面这段代码是使用内置装置的情形：

msgqueue: Message queue broker in C

```
//
//  简单消息队列代理
//  功能和请求-应答代理相同，但使用了内置的装置
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  客户端套接字
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (frontend, "tcp://*:5559");

    //  服务端套接字
    void *backend = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (backend, "tcp://*:5560");

    //  启动内置装置
    zmq_device (ZMQ_QUEUE, frontend, backend);

    //  程序不会运行到这里
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

内置装置会恰当地处理错误，而我们手工实现的代理并没有加入错误处理机制。所以说，当你能够在程序中使用内置装置的时候就尽量用吧。

可能你会像某些ZMQ开发者一样提出这样一个问题：如果我将其他类型的套接字传入这些装置中会发生什么？答案是：别这么做。你可以随意传入不同类型的套接字，但是执行结果会非常奇怪。所以，QUEUE装置应使用ROUTER/DEALER套接字、FORWARDER应使用SUB/PUB、STREAMER应使用PULL/PUSH。

当你需要其他的套接字类型进行组合时，那就需要自己编写装置了。

ZMQ多线程编程

使用ZMQ进行多线程编程（MT编程）将会是一种享受。在多线程中使用ZMQ套接字时，你不需要考虑额外的东西，让它们自如地运作就好。

使用ZMQ进行多线程编程时，不需要考虑互斥、锁、或其他并发程序中要考虑的因素，你唯一要关心的仅仅是线程之间的消息。

什么叫“完美”的多线程编程，指的是代码易写易读，可以跨系统、跨语言地使用同一种技术，能够在任意颗核心的计算机上运行，没有状态，没有速度的瓶颈。

如果你有多年的多线程编程经验，知道如何使用锁、信号灯、临界区等机制来使代码运行得正确（尚未考虑快速），那你可能会很沮丧，因为ZMQ将改变这一切。三十多年来，并发式应用程序开发所总结的经验是：不要共享状态。这就好比两个醉汉想要分享一杯啤酒，如果他们不是铁哥们儿，那他们很快就会打起来。当有更多的醉汉加入时，情况就会更糟。多线程编程有时就像醉汉抢夺啤酒那样糟糕。

进行多线程编程往往是痛苦的，当程序因为压力过大而崩溃时，你会不知所然。有人写过一篇《多线程代码中的11个错误易发点》的文章，在大公司中广为流传，列举其中的几项：没有进行同步、错误的粒度、读写分离、无锁排序、锁传递、优先级冲突等。

假设某一天的下午三点，当证券市场正交易得如火如荼的时候，突然之间，应用程序因为锁的问题崩溃了，那将会是何等的场景？所以，作为程序员的我们，为解决那些复杂的多线程问题，只能用上更复杂的编程机制。

有人曾这样比喻，那些多线程程序原本应作为大型公司的核心支柱，但往往又最容易出错；那些想要通过网络不断进行延伸的产品，最后总以失败告终。

如何用ZMQ进行多线程编程，以下是一些规则：

- 不要在不同的线程之间访问同一份数据，如果要用到传统编程中的互斥机制，那就有违ZMQ的思想了。唯一的例外是ZMQ上下文对象，它是线程安全的。
- 必须为进程创建ZMQ上下文，并将其传递给所有你需要使用inproc协议进行通信的线程；
- 你可以将线程作为单独的任务来对待，使用自己的上下文，但是这些线程之间就不能使用inproc协议进行通信了。这样做的好处是可以在日后方便地将程序拆分为不同的进程来运行。
- 不要在不同的线程之间传递套接字对象，这些对象不是线程安全的。从技术上来说，你是可以这样做的，但是会用到互斥和锁的机制，这会让你的应用程序变得缓慢和脆弱。唯一合理的情形是，在某些语言的ZMQ类库内部，需要使用垃圾回收机制，这时可能会进行套接字对象的传递。

当你需要在应用程序中使用两个装置时，可能会将套接字对象从一个线程传递给另一个线程，这样做一开始可能会成功，但最后一定会随机地发生错误。所以说，应在同一个线程中打开和关闭套接字。

如果你能遵循上面的规则，就会发现多线程程序可以很容易地拆分成多个进程。程序逻辑可以在线程、进程、或是计算机中运行，根据你的需求进行部署即可。

ZMQ使用的是系统原生的线程机制，而不是某种“绿色线程”。这样做的好处是你不需要学习新的多线程编程API，而且可以和目标操作系统进行很好的结合。你可以使用类似英特尔的ThreadChecker工具来查看线程工作的情况。缺点在于，如果程序创建了太多的线程（如上千个），则可能导致操作系统负载过高。

下面我们举一个实例，让原来的Hello World服务变得更为强大。原来的服务是单线程的，如果请求较少，自然没有问题。ZMQ的线程可以在一个核心上高速地运行，执行大量的工作。但是，如果有一万次请求同时发送过来会怎么样？因此，现实环境中，我们会启动多个worker线程，他们会尽可能地接收客户端请求，处理并返回应答。

当然，我们可以使用启动多个worker进程的方式来实现，但是启动一个进程总比启动多个进程要来的方便且易于管理。而且，作为线程启动的worker，所占用的带宽会比较少，延迟也会较低。 以下是多线程版的Hello World服务：

mtserver: Multithreaded service in C

```
//
//  多线程版Hello World服务
//
#include "zhelpers.h"
#include <pthread.h>

static void *
worker_routine (void *context) {
    //  连接至代理的套接字
    void *receiver = zmq_socket (context, ZMQ_REP);
    zmq_connect (receiver, "inproc://workers");

    while (1) {
        char *string = s_recv (receiver);
        printf ("Received request: [%s]\n", string);
        free (string);
        //  工作
        sleep (1);
        //  返回应答
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}

int main (void)
{
    void *context = zmq_init (1);

    //  用于和client进行通信的套接字
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "tcp://*:5555");

    //  用于和worker进行通信的套接字
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

    //  启动一个worker池
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
```

```
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    //  启动队列装置
    zmq_device (ZMQ_QUEUE, clients, workers);

    //  程序不会运行到这里，但仍进行清理工作
    zmq_close (clients);
    zmq_close (workers);
    zmq_term (context);
    return 0;
}
```

所有的代码应该都已经很熟悉了：

- 服务端启动一组worker线程，每个worker创建一个REP套接字，并处理收到的请求。worker线程就像是一个单线程的服务，唯一的区别是使用了inproc而非tcp协议，以及绑定-连接的方向调换了。
- 服务端创建ROUTER套接字用以和client通信，因此提供了一个TCP协议的外部接口。
- 服务端创建DEALER套接字用以和worker通信，使用了内部接口（inproc）。
- 服务端启动了QUEUE内部装置，连接两个端点上的套接字。QUEUE装置会将收到的请求分发给连接上的worker，并将应答路由给请求方。

需要注意的是，在某些编程语言中，创建线程并不是特别方便，POSIX提供的类库是pthreads，但Windows中就需要使用不同的API了。我们会在第三章中讲述如何包装一个多线程编程的API。

示例中的“工作”仅仅是1秒钟的停留，我们可以在worker中进行任意的操作，包括与其他节点进行通信。消息的流向是这样的：REQ-ROUTER-queue-DEALER-REP。

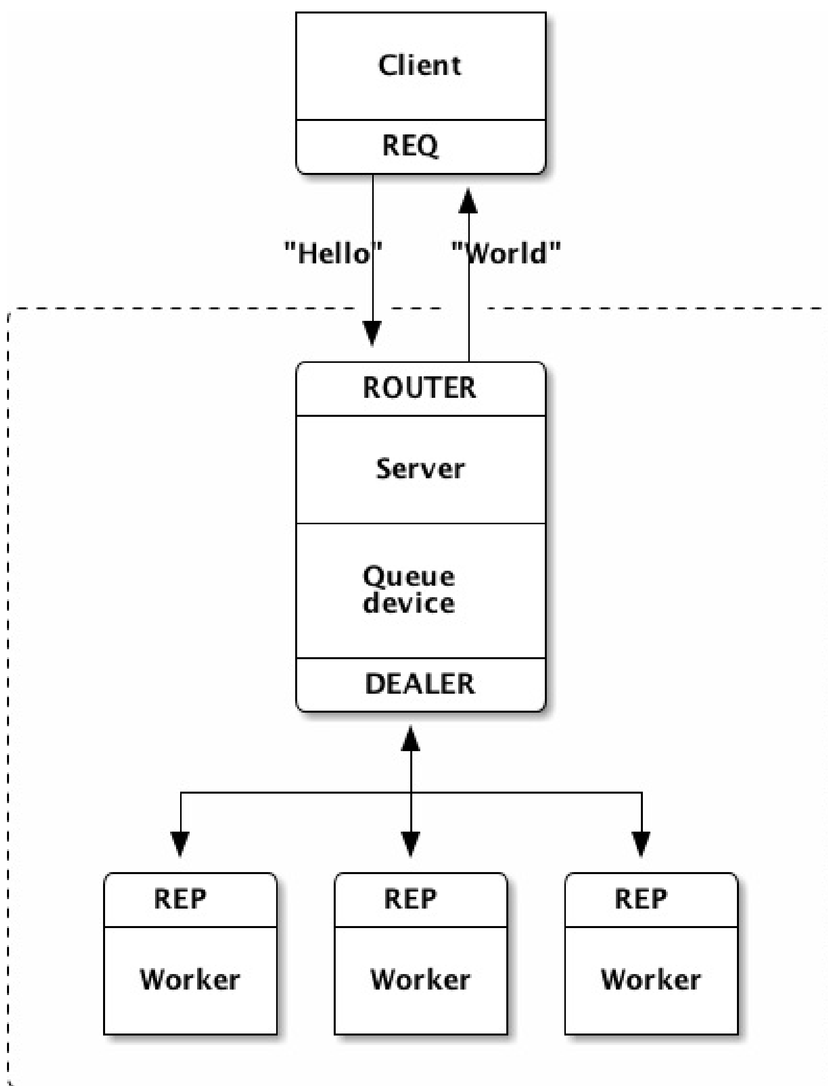


Figure 12 — Multithreaded server

线程间的信号传输

当你刚开始使用ZMQ进行多线程编程时，你可能会问：要如何协调两个线程的工作呢？可能会想要使用sleep()这样的方法，或者使用诸如信号、互斥等机制。事实上，你唯一要用的就是**ZMQ**本身。回忆一下那个醉汉抢啤酒的例子吧。

下面的示例演示了三个线程之间需要如何进行同步：

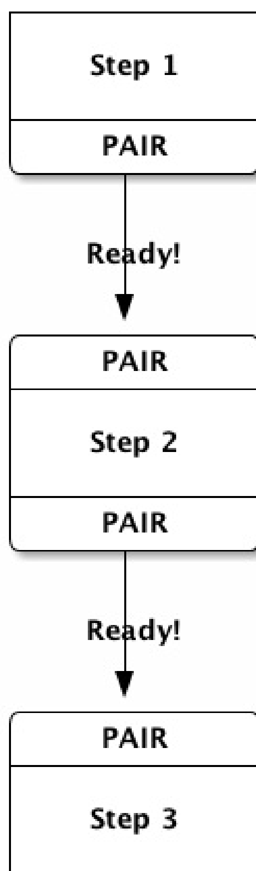


Figure 13 — The Relay Race

我们使用PAIR套接字和inproc协议。

mtrelay: Multithreaded relay in C

```
//
// 多线程同步
//
#include "zhelpers.h"
#include <pthread.h>

static void *
step1 (void *context) {
    // 连接至步骤2，告知我已就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step2");
    printf ("步骤1就绪，正在通知步骤2……\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

static void *
step2 (void *context) {
    // 启动步骤1前线绑定至inproc套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
```

```
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    //  等待信号
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    //  连接至步骤3，告知我已就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step3");
    printf ("步骤2就绪，正在通知步骤3……\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

int main (void)
{
    void *context = zmq_init (1);

    //  启动步骤2前线绑定至inproc套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    //  等待信号
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    printf ("测试成功！\n");
    zmq_term (context);
    return 0;
}
```

这是一个ZMQ多线程编程的典型示例：

1. 两个线程通过inproc协议进行通信，使用同一个上下文；
2. 父线程创建一个套接字，绑定至inproc://端点，然后再启动子线程，将上下文对象传递给它；
3. 子线程创建第二个套接字，连接至inproc://端点，然后发送已就绪信号给父线程。

需要注意的是，这段代码无法扩展到多个进程之间的协调。如果你使用inproc协议，只能建立结构非常紧密的应用程序。在延迟时间必须严格控制的情况下可以使用这种方法。对其他应用程序来说，每个线程使用同一个上下文，协议选用ipc或tcp。然后，你就可以自由地将应用程序拆分为多个进程甚至是多台计算机了。

这是我们第一次使用PAIR套接字。为什么要使用PAIR？其他类型的套接字也可以使用，但都有一些缺点会影响到线程间的通信：

- 你可以让信号发送方使用PUSH，接收方使用PULL，这看上去可能可以，但是需要注意的是，PUSH套接字发送消息时会进行负载均衡，如果你不小心开启了两个接收方，就会“丢失”一半的信号。而PAIR套接字建立的是一对一的连接，具有排他性。

- 可以让发送方使用DEALER，接收方使用ROUTER。但是，ROUTER套接字会在消息的外层包裹一个来源地址，这样一来原本零字节的信号就可能要成为一个多段消息了。如果你不在乎这个问题，并且不会重复读取那个套接字，自然可以使用这种方法。但是，如果你想要使用这个套接字接收真正的数据，你就会发现ROUTER提供的消息是错误的。至于DEALER套接字，它同样有负载均衡的机制，和PUSH套接字有相同的风险。

- 可以让发送方使用PUB，接收方使用SUB。一来消息可以照原样发送，二来PUB套接字不会进行负载均衡。但是，你需要对SUB套接字设置一个空的订阅信息（用以接收所有消息）；而且，如果SUB套接字没有及时和PUB建立连接，消息很有可能会丢失。

综上，使用PAIR套接字进行线程间的协调是最合适的。

节点协调

当你想要对节点进行协调时，PAIR套接字就不怎么合适了，这也是线程和节点之间的不同之处。一般来说，节点是来去自由的，而线程则较为稳定。使用PAIR套接字时，若远程节点断开连接后又进行重连，PAIR不会予以理会。

第二个区别在于，线程的数量一般是固定的，而节点数量则会经常变化。让我们以气象信息模型为基础，看看要怎样进行节点的协调，以保证客户端不会丢失最开始的那些消息。

下面是程序运行逻辑：

- 发布者知道预期的订阅者数量，这个数字可以任意指定；
- 发布者启动后会先等待所有订阅者进行连接，也就是节点协调。每个订阅者会使用另一个套接字来告知发布者自己已就绪；
- 当所有订阅者准备就绪后，发布者才开始发送消息。

这里我们会使用REQ-REP套接字来同步发布者和订阅者。发布者的代码如下：

syncpub: Synchronized publisher in C

```
//
//  发布者 - 同步版
//
#include "zhelpers.h"

//  等待10个订阅者连接
#define SUBSCRIBERS_EXPECTED  10

int main (void)
{
    void *context = zmq_init (1);

    //  用于和客户端通信的套接字
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5561");

    //  用于接收信号的套接字
    void *syncservice = zmq_socket (context, ZMQ_REP);
    zmq_bind (syncservice, "tcp://*:5562");

    //  接收订阅者的就绪信号
    printf ("正在等待订阅者就绪\n");
    int subscribers = 0;
    while (subscribers < SUBSCRIBERS_EXPECTED) {
        //  - 等待就绪信息
        char *string = s_recv (syncservice);
        free (string);
        //  - 发送应答
        s_send (syncservice, "");
        subscribers++;
    }
    //  开始发送100万条数据
    printf ("正在广播消息\n");
    int update_nbr;
    for (update_nbr = 0; update_nbr < 1000000; update_nbr++)
        s_send (publisher, "Rhubarb");

    s_send (publisher, "END");

    zmq_close (publisher);
    zmq_close (syncservice);
    zmq_term (context);
    return 0;
}
```

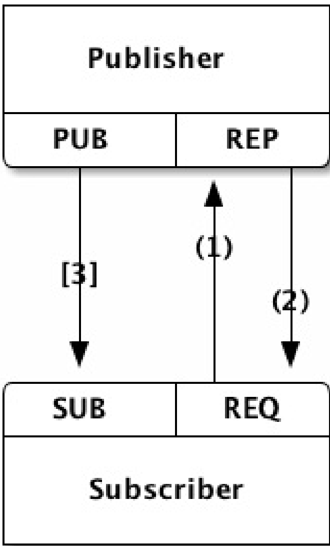


Figure 14 — Pub Sub Synchronization

以下是订阅者的代码：

syncsub: Synchronized subscriber in C

```
//
//  订阅者 - 同步版
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  一、连接SUB套接字
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5561");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

    //  ZMQ太快了，我们延迟一会儿……
    sleep (1);

    //  二、与发布者进行同步
    void *syncclient = zmq_socket (context, ZMQ_REQ);
    zmq_connect (syncclient, "tcp://localhost:5562");

    //  - 发送请求
    s_send (syncclient, "");

    //  - 等待应答
    char *string = s_recv (syncclient);
    free (string);

    //  三、处理消息
    int update_nbr = 0;
    while (1) {
        char *string = s_recv (subscriber);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
        update_nbr++;
    }
    printf ("收到 %d 条消息\n", update_nbr);

    zmq_close (subscriber);
    zmq_close (syncclient);
    zmq_term (context);
    return 0;
}
```

以下这段shell脚本会启动10个订阅者、1个发布者：

```
echo "正在启动订阅者..."
for a in 1 2 3 4 5 6 7 8 9 10; do
    syncsub &
done
echo "正在启动发布者..."
syncpub
```

结果如下：

```
正在启动订阅者...
正在启动发布者...
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
```

当REQ-REP请求完成时，我们仍无法保证SUB套接字已成功建立连接。除非使用inproc协议，否则对外连接的顺序是不一定的。因此，示例程序中使用了sleep(1)的方式来进行处理，随后再发送同步请求。

更可靠的模型可以是：

- 发布者打开PUB套接字，开始发送Hello消息（非数据）；
- 订阅者连接SUB套接字，当收到Hello消息后再使用REQ-REP套接字进行同步；
- 当发布者获得所有订阅者的同步消息后，才开始发送真正的数据。

零拷贝

第一章中我们曾提过零拷贝是很危险的，其实那是吓唬你的。既然你已经读到这里了，说明你已经具备了足够的知识，能够使用零拷贝。但需要记住，条条大路通地狱，过早地对程序进行优化其实是没有必要的。简单的说，如果你用不好零拷贝，那可能会让程序架构变得更糟。

ZMQ提供的API可以让你直接发送和接收消息，不用考虑缓存的问题。正因为消息是由ZMQ在后台收发的，所以使用零拷贝需要一些额外的工作。

做零拷贝时，使用zmq_msg_init_data()函数创建一条消息，其内容指向某个已经分配好的内存区域，然后将该消息传递给zmq_send()函数。创建消息时，你还需要提供一个用于释放消息内容的函数，ZMQ会在消息发送完毕时调用。下面是一个简单的例子，我们假设已经分配好的内存区域为1000个字节：

```
void my_free (void *data, void *hint) {
    free (data);
}
//  Send message from buffer, which we allocate and 0MQ will free f
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_send (socket, &message, 0);
```

在接收消息的时候是无法使用零拷贝的：ZMQ会将收到的消息放入一块内存区域供你读取，但不会将消息写入程序指定的内存区域。

ZMQ的多段消息能够很好地支持零拷贝。在传统消息系统中，你需要将不同缓存中的内容保存到同一个缓存中，然后才能发送。但ZMQ会将来自不同内存区域的内容作为消息的一个帧进行发送。而且在ZMQ内部，一条消息会作为一个整体进行收发，因而非常高效。

瞬时套接字和持久套接字

在传统网络编程中，套接字是一个API对象，它们的生命周期不会长过程序的生命周期。但仔细打量一下套接字，它会占用一项特定的资源——缓存，这时ZMQ的开发者可能会问：是否有办法在程序崩溃时让这些套接字缓存得以保留，稍后能够恢复？

这种特性应该会非常有用，虽然不能应对所有的危险，但至少可以挽回一部分损失，特别是多发布-订阅模式来说。让我们来讨论一下。

这里有两个套接字正在欢快地传送着气象信息：

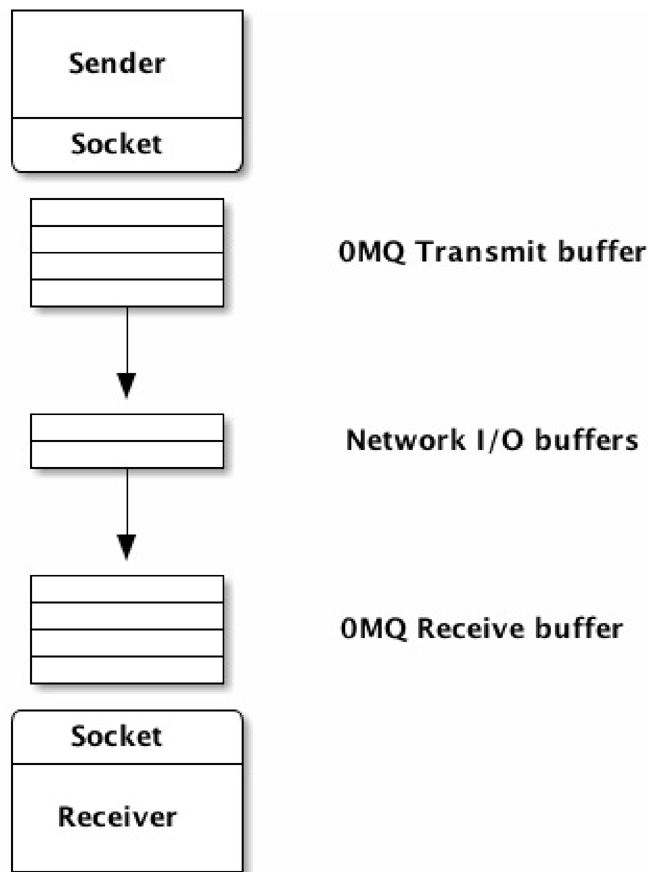


Figure 15 — Sender boring the pants off receiver

如果接收方（SUB、PULL、REQ）指定了套接字标识，当它们断开网络时，发送方（PUB、PUSH、REP）会为它们缓存信息，直至达到阈值（HWM）。这里发送方不需要有套接字标识。

需要注意，ZMQ的套接字缓存对程序原来说是不可见的，正如TCP缓存一样。

到目前为止，我们使用的套接字都是瞬时套接字。要将瞬时套接字转化为持久套接字，需要为其设定一个套接字标识。所有的ZMQ套接字都会有一个标识，不过是由ZMQ自动生成的UUID。

在ZMQ内部，两个套接字相连时会先交换各自的标识。如果发生对方没有ID，则会自行生成一个用以标识对方：

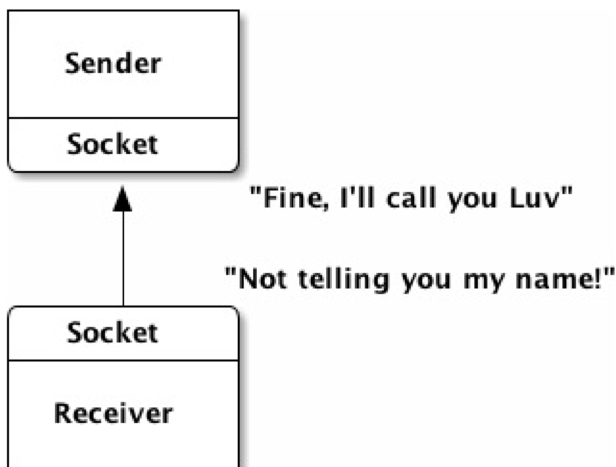


Figure 16 — Transient socket

但套接字也可以告知对方自己的标识，那当它们第二次连接时，就能知道对方的身份：

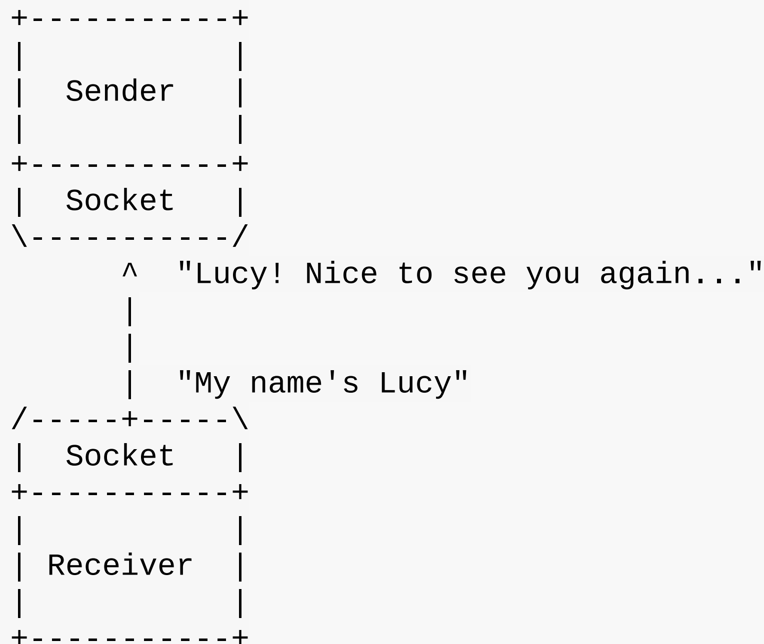


Figure # - Durable socket

下面这行代码就可以为套接字设置标识，从而建立了一个持久的套接字：

```
zmq_setsockopt (socket, ZMQ_IDENTITY, "Lucy", 4);
```

关于套接字标识还有几点说明：

- 如果要为套接字设置标识，必须在连接或绑定至端点之前设置；
- 接收方会选择使用套接字标识，正如cookie在HTTP网页应用中的性质，是由服务器去选择要使用哪个cookie的；

- 套接字标识是二进制字符串；以字节0开头的套接字标识为ZMQ保留标识；
- 不用为多个套接字指定相同的标识，若套接字使用的标识已被占用，它将无法连接至其他套接字；
- 不要使用随机的套接字标识，这样会生成很多持久化套接字，最终让节点崩溃；
- 如果你想获取对方套接字的标识，只有ROUTER套接字会帮你自动完成这件事，使用其他套接字类型时，需要将标识作为消息的一帧发送过来；
- 说了以上这些，使用持久化套接字其实并不明智，因为它会让发送者越来越混乱，让架构变得脆弱。如果我们能重新设计ZMQ，很可能会去掉这种显式声明套接字标识的功能。

其他信息可以查看zmq_setsockopt()函数的ZMQ_IDENTITY一节。注意，该方法只能获取程序中套接字的标识，而不能获得对方套接字的标识。

发布-订阅消息信封

我们简单介绍了多帧消息，下面就来看看它的典型用法——消息信封。信封是指为消息注明来源地址，而不修改消息内容。

在发布-订阅模式中，信封包含了订阅信息，用以过滤掉不需要接收的消息。

如果你想要使用发布-订阅信封，就需要自行生成和设置。这个动作是可选的，我们在之前的示例中也没有使用到。在发布-订阅模式中使用信封可能会比较麻烦，但在现实应用中还是很有必要的，毕竟信封和消息的确是两块不想干的数据。

这是发布-订阅模式中一个带有信封的消息：

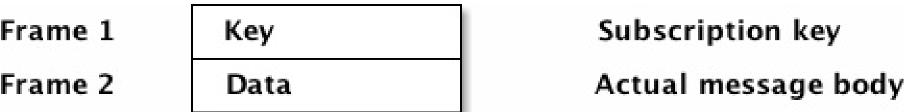


Figure 17 – Pub sub envelope with separate key

我们回忆一下，发布-订阅模式中，消息的接收是根据订阅信息来的，也就是消息的前缀。将这个前缀放入单独的消息帧，可以让匹配变得非常明显。因为不会有一个应用程序恰好只匹配了一部分数据。

下面是一个最简的发布-订阅消息信封示例。发布者会发送两类消息：A和B，信封中指明了消息类型：

psenvpub: Pub-sub envelope publisher in C

```
//
//  发布-订阅消息信封 - 发布者
//  s_sendmore()函数也是zhelpers.h提供的
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和PUB套接字
    void *context = zmq_init (1);
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5563");

    while (1) {
        //  发布两条消息，A类型和B类型
        s_sendmore (publisher, "A");
        s_send (publisher, "We don't want to see this");
        s_sendmore (publisher, "B");
        s_send (publisher, "We would like to see this");
        sleep (1);
    }
    //  正确退出
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

假设订阅者只需要B类型的消息：

psenvsub: Pub-sub envelope subscriber in C

```
//
//  发布-订阅消息信封 - 订阅者
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和SUB套接字
    void *context = zmq_init (1);
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5563");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "B", 1);

    while (1) {
        //  读取消息信封
        char *address = s_recv (subscriber);
        //  读取消息内容
        char *contents = s_recv (subscriber);
        printf ("[%s] %s\n", address, contents);
        free (address);
        free (contents);
    }
    //  正确退出
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

执行上面的程序时，订阅者会打印如下信息：

```
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
...
```

这个示例说明订阅者会丢弃未订阅的消息，且接收完整的多帧消息——你不会只获得消息的一部分。

如果你订阅了多个套接字，又想知道这些套接字的标识，从而通过另一个套接字来发送消息给它们（这个用例很常见），你可以让发布者创建一条含有三帧的消息：

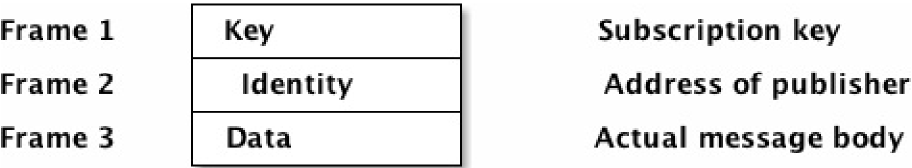


Figure 18 – Pub sub envelope with sender address

（半）持久订阅者和阈值（HWM）

所有的套接字类型都可以使用标识。如果你在使用PUB和SUB套接字，其中SUB套接字为自己声明了标识，那么，当SUB断开连接时，PUB会保留要发送给SUB的消息。

这种机制有好有坏。好的地方在于发布者会暂存这些消息，当订阅者重连后进行发送；不好的地方在于这样很容易让发布者因内存溢出而崩溃。

如果你在使用持久化的**SUB**套接字（即为**SUB**设置了套接字标识），那么你必须设法避免消息在发布者队列中堆砌并溢出，应该使用阈值（**HWM**）来保护发布者套接字。发布者的阈值会分别影响所有的订阅者。

我们可以运行一个示例来证明这一点，用第一章中的wuclient和wuserver具体，在wuclient中进行套接字连接前加入这一行：

```
zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);
```

编译并运行这两段程序，一切看起来都很平常。但是观察一下发布者的内存占用情况，可以看到当订阅者逐个退出后，发布者的内存占用会逐渐上升。若此时你重启订阅者，会发现发布者的内存占用不再增长了，一旦订阅者停止，就又会增长。很快地，它就会耗尽系统资源。

我们先来看看如何设置阈值，然后再看如何设置得正确。下面的发布者和订阅者使用了上文提到的“节点协调”机制。发布者会每隔一秒发送一条消息，这时你可以中断订阅者，重新启动它，看看会发生什么。

以下是发布者的代码：

durapub: Durable publisher in C

```
//
//  发布者 - 连接持久化的订阅者
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  订阅者会发送已就绪的消息
    void *sync = zmq_socket (context, ZMQ_PULL);
    zmq_bind (sync, "tcp://*:5564");

    //  使用该套接字发布消息
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5565");

    //  等待同步消息
    char *string = s_recv (sync);
    free (string);

    //  广播10条消息，一秒一条
    int update_nbr;
    for (update_nbr = 0; update_nbr < 10; update_nbr++) {
        char string [20];
        sprintf (string, "Update %d", update_nbr);
        s_send (publisher, string);
        sleep (1);
    }
    s_send (publisher, "END");

    zmq_close (sync);
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

下面是订阅者的代码：

durasub: Durable subscriber in C

```
//
//  持久化的订阅者
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  连接SUB套接字
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);
    zmq_connect (subscriber, "tcp://localhost:5565");

    //  发送同步消息
    void *sync = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sync, "tcp://localhost:5564");
    s_send (sync, "");

    //  获取更新，并按指令退出
    while (1) {
        char *string = s_recv (subscriber);
        printf ("%s\n", string);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
    }
    zmq_close (sync);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

运行以上代码，在不同的窗口中先后打开发布者和订阅者。当订阅者获取了一至两条消息后按Ctrl-C中止，然后重新启动，看看执行结果：

```
$ durasub
Update 0
Update 1
Update 2
^C
$ durasub
Update 3
Update 4
Update 5
Update 6
Update 7
^C
$ durasub
Update 8
Update 9
END
```

可以看到订阅者的唯一区别是为套接字设置了标识，发布者就会将消息缓存起来，待重建连接后发送。设置套接字标识可以让瞬时套接字转变为持久套接字。实践中，你需要小心地给套接字起名字，可以从配置文件中获取，或者生成一个UUID并保存起来。

当我们为PUB套接字设置了阈值，发布者就会缓存指定数量的消息，转而丢弃溢出的消息。让我们将阈值设置为2，看看会发生什么：

```
uint64_t hwm = 2;
zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));
```

运行程序，中断订阅者后等待一段时间再重启，可以看到结果如下：

```
$ durasub
Update 0
Update 1
^C
$ durasub
Update 2
Update 3
Update 7
Update 8
Update 9
END
```

看仔细了，发布者只为我们保存了两条消息（2和3）。阈值使得ZMQ丢弃溢出队列的消息。

简而言之，如果你要使用持久化的订阅者，就必须在发布者端设置阈值，否则可能造成服务器因内存溢出而崩溃。但是，还有另一种方法。ZMQ提供了名为交换区（swap）的机制，它是一个磁盘文件，用于存放从队列中溢出的消息。启动它很简单：

```
// 指定交换区大小，单位：字节。
uint64_t swap = 25000000;
zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof (swap));
```

我们可以将上面的方法综合起来，编写一个既能接受持久化套接字，又不至于内存溢出的发布者：

durapub2: Durable but cynical publisher in C

```
//
//  发布者 - 连接持久化订阅者
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    //  订阅者会告知我们它已就绪
    void *sync = zmq_socket (context, ZMQ_PULL);
    zmq_bind (sync, "tcp://*:5564");

    //  使用该套接字发送消息
    void *publisher = zmq_socket (context, ZMQ_PUB);

    //  避免慢持久化订阅者消息溢出的问题
    uint64_t hwm = 1;
    zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));

    //  设置交换区大小，供所有订阅者使用
    uint64_t swap = 25000000;
    zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof (swap));
    zmq_bind (publisher, "tcp://*:5565");

    //  等待同步消息
    char *string = s_recv (sync);
    free (string);

    //  发布10条消息，一秒一条
    int update_nbr;
    for (update_nbr = 0; update_nbr < 10; update_nbr++) {
        char string [20];
        sprintf (string, "Update %d", update_nbr);
        s_send (publisher, string);
        sleep (1);
    }
    s_send (publisher, "END");

    zmq_close (sync);
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

若在现实环境中将阈值设置为1，致使所有待发送的消息都保存到磁盘上，会大大降低处理速度。这里有一些典型的方法用以处理不同的订阅者：

- 必须为PUB套接字设置阈值，具体数字可以通过最大订阅者数、可供队列使用的最大内存区域、以及消息的平均大小来衡量。举例来说，你预计会有5000个订阅者，有1G的内存可供使用，消息大小在200个字节左右，那么，一个合理

的阈值是1,000,000,000 / 200 / 5,000 = 1,000。

- 如果你不希望慢速或崩溃的订阅者丢失消息，可以设置一个交换区，在高峰期的时候存放这些消息。交换区的大小可以根据订阅者数、高峰消息比率、消息平均大小、暂存时间等来衡量。比如，你预计有5000个订阅者，消息大小为200个字节左右，每秒会有10万条消息。这样，你每秒就需要100MB的磁盘空间来存放消息。加总起来，你会需要6GB的磁盘空间，而且必须足够的快（这超出了本指南的讲解范围）。

关于持久化订阅者：

- 数据可能会丢失，这要看消息发布的频率、网络缓存大小、通信协议等。持久化的订阅者比起瞬时套接字要可靠一些，但也并不是完美的。

- 交换区文件是无法恢复的，所以当发布者或代理消亡时，交换区中的数据仍然会丢失。

关于阈值：

- 这个选项会同时影响套接字的发送和接收队列。当然，PUB、PUSH不会有接收队列，SUB、PULL、REQ、REP不会有发送队列。而像DEALER、ROUTER、PAIR套接字时，他们既有发送队列，又有接收队列。

- 当套接字达到阈值时，ZMQ会发生阻塞，或直接丢弃消息。

- 使用inproc协议时，发送者和接受者共享同一个队列缓存，所以说，真正的阈值是两个套接字阈值之和。如果一方套接字没有设置阈值，那么它就不会有缓存方面的限制。

这就是你想要的！

ZMQ就像是一盒积木，只要你有足够的想象力，就可以用它组装出任何造型的网络架构。

这种高可扩、高弹性的架构一定会打开你的眼界。其实这并不是ZMQ原创的，早就有像Erlang这样的基于流的编程语言已经能够做到了，只是ZMQ提供了更为友善和易用的接口。

正如Gonzo Diethelm所言：“我想用一句话来总结，‘如果ZMQ不存在，那它就应该被发明出来。’作为一个有着多年相关工作经验的人，ZMQ太能引起我的共鸣了。我只能说，‘这就是我想要的！’”

第三章 高级请求-应答模式

在第二章中我们通过开发一系列的小应用来熟悉ØMQ的基本使用方法，每个应用会引入一些新的特性。本章会沿用这种方式，来探索更多建立在ØMQ请求-应答模式之上的高级工作模式。

本章涉及的内容有：

- 在请求-应答模式中创建和使用消息信封
- 使用REQ、REP、DEALER和ROUTER套接字
- 使用标识来手工指定应答目标
- 使用自定义离散路由模式
- 使用自定义最近最少使用路由模式
- 构建高层消息封装类
- 构建基本的请求应答代理
- 合理命名套接字
- 模拟client-worker集群
- 构建可扩展的请求-应答集群云
- 使用管道套接字监控线程

Request-Reply Envelopes

在请求-应答模式中，信封里保存了应答目标的位置。这就是为什么ØMQ网络虽然是无状态的，但仍能完成请求-应答的过程。

在一般使用过程中，你并不需要知道请求-应答信封的工作原理。使用REQ、REP时，ØMQ会自动处理消息信封。下一章讲到的装置（device），使用时也只需保证读取和写入所有的信息即可。ØMQ使用多段消息的方式来存储信封，所以在复制消息时也会复制信封。

然而，在使用高级请求-应答模式之前是需要了解信封这一机制的，以下是信封机制在ROUTER中的工作原理：

- 从ROUTER中读取一条消息时，ØMQ会包上一层信封，上面注明了消息的来源。
- 向ROUTER写入一条消息时（包含信封），ØMQ会将信封拆开，并将消息递送给相应的对象。

如果将从ROUTER A中获取的消息（包含信封）写入ROUTER B（即将消息发送给一个DEALER，该DEALER连接到了ROUTER），那么在从ROUTER B中获取该消息时就会包含两层信封。

信封机制的根本作用是让ROUTER知道如何将消息递送给正确的应答目标，你需要做的就是在程序中保留好该信封。回顾一下REP套接字，它会将收到消息的信封逐个拆开，将消息本身传送给应用程序。而在发送时，又会在消息外层包裹该信封，发送给ROUTER，从而传递给正确的应答目标。

我们可以使用上述原理建立起一个ROUTER-DEALER装置：

```
[REQ] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [ROUTER--DEALER] <--> [REP]
...etc.
```

当你用REQ套接字去连接ROUTER套接字，并发送一条请求消息，你会从ROUTER中获得一条如下所示的消息：



Figure 1 — Single hop request-reply envelope

- 第三帧是应用程序发送给REQ套接字的消息；
- 第二帧的空信息是REQ套接字在发送消息给ROUTER之前添加的；
- 第一帧即信封，是由ROUTER套接字添加的，记录了消息的来源。

如果我们在一条装置链路上传递该消息，最终会得到包含多层信封的消息。最新的信封会在消息的顶部。

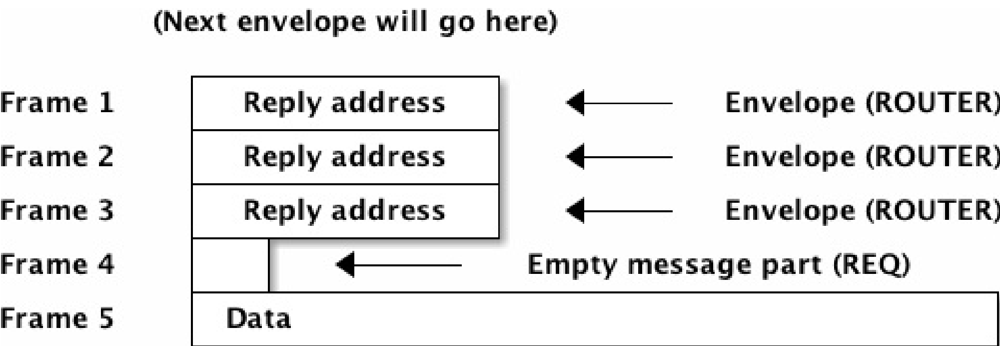


Figure 2 — Multihop request-reply envelope

以下将详述我们在请求-应答模式中使用到的四种套接字类型：

- DEALER是一种负载均衡，它会将消息分发给已连接的节点，并使用公平队列的机制处理接受到的消息。DEALER的作用就像是PUSH和PULL的结合。

- REQ发送消息时会在消息顶部插入一个空帧，接受时会将空帧移去。其实REQ是建立在DEALER之上的，但REQ只有当消息发送并接受到回应后才能继续运行。

- ROUTER在收到消息时会在顶部添加一个信封，标记消息来源。发送时会通过该信封决定哪个节点可以获取到该条消息。

- REP在收到消息时会将第一个空帧之前的所有信息保存起来，将原始信息传送给应用程序。在发送消息时，REP会用刚才保存的信息包裹应答消息。REP其实是建立在ROUTER之上的，但和REQ一样，必须完成接受和发送这两个动作

后才能继续。

REP要求消息中的信封由一个空帧结束，所以如果你没有用REQ发送消息，则需要自己在消息中添加这个空帧。

你肯定会问，ROUTER是怎么标识消息的来源的？答案当然是套接字的标识。我们之前讲过，一个套接字可能是瞬时的，它所连接的套接字（如ROUTER）则会给它生成一个标识，与之相关联。一个套接字也可以显式地给自己定义一个标识，这样其他套接字就可以直接使用了。

这是一个瞬时的套接字，ROUTER会自动生成一个UUID来标识消息的来源。

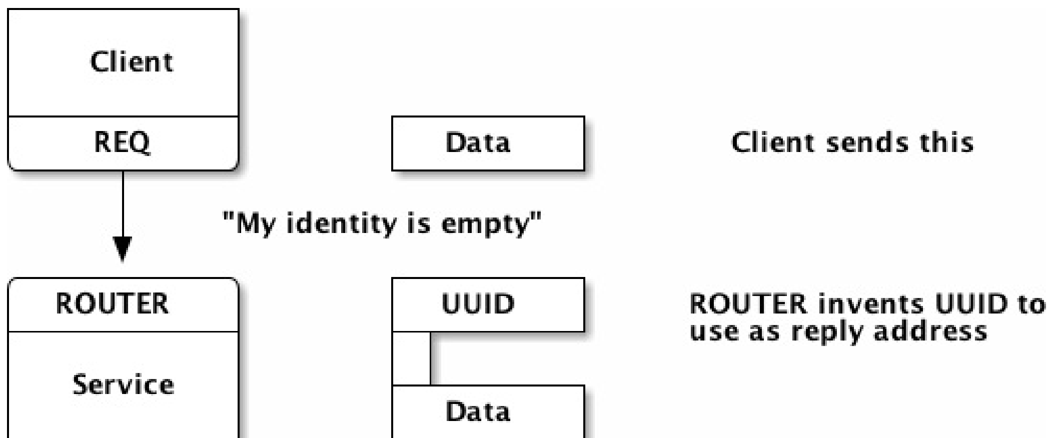


Figure 3 — ROUTER invents a UUID for transient sockets

这是一个持久的套接字，标识由消息来源自己指定。

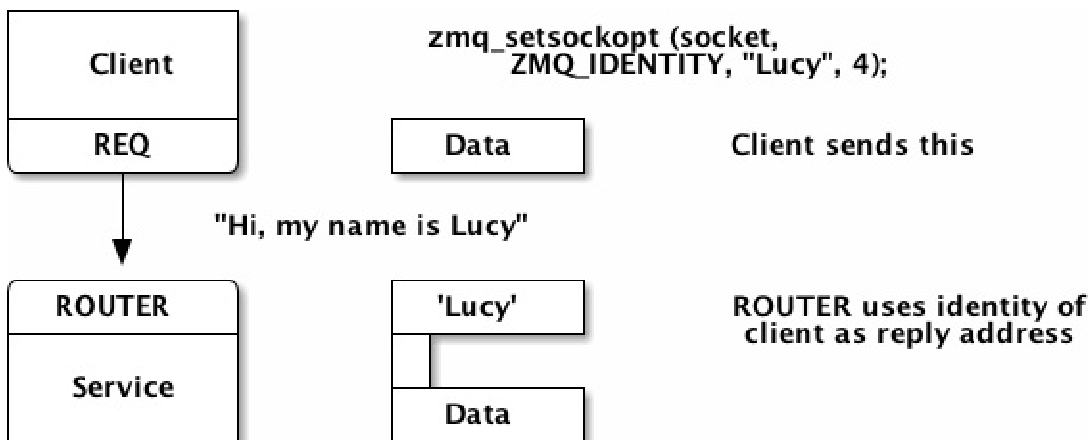


Figure 4 — ROUTER uses identity if it knows it

下面让我们在实例中观察上述两种操作。下列程序会打印出ROUTER从两个REP套接字中获得的消息，其中一个没有指定标识，另一个指定了“Hello”作为标识。

identity.c

```
//
// 以下程序演示了如何在请求-应答模式中使用套接字标识。
// 需要注意的是s_开头的函数是由zhelpers.h提供的。
// 我们没有必要重复编写那些代码。
//
#include "zhelpers.h"

int main (void)
{
    void *context = zmq_init (1);

    void *sink = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (sink, "inproc://example");

    // 第一个套接字由0MQ自动设置标识
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous, "ROUTER uses a generated UUID");
    s_dump (sink);

    // 第二个由自己设置
    void *identified = zmq_socket (context, ZMQ_REQ);
    zmq_setsockopt (identified, ZMQ_IDENTITY, "Hello", 5);
    zmq_connect (identified, "inproc://example");
    s_send (identified, "ROUTER socket uses REQ's socket identity")
    s_dump (sink);

    zmq_close (sink);
    zmq_close (anonymous);
    zmq_close (identified);
    zmq_term (context);
    return 0;
}
```

运行结果：

```
----------------------------------------
[017] 00314F043F46C441E28DD0AC54BE8DA727
[000]
[026] ROUTER uses a generated UUID
----------------------------------------
[005] Hello
[000]
[038] ROUTER socket uses REQ's socket identity
```

自定义请求-应答路由

我们已经看到ROUTER套接字是如何使用信封将消息发送给正确的应答目标的，下面我们从一个角度来定义ROUTER：在发送消息时使用一定格式的信封提供正确的路由目标，ROUTER就能够将该条消息异步地发送给对应的节点。

所以说ROUTER的行为是完全可控的。在深入理解这一特性之前，让我们先近距离观察一下REQ和REP套接字，赋予他们一些鲜活的角色：

- REQ是一个“妈妈”套接字，不会耐心听别人说话，但会不断地抛出问题寻求解答。REQ是严格同步的，它永远位于消息链路的请求端；
- REP则是一个“爸爸”套接字，只会回答问题，不会主动和别人对话。REP也是严格同步的，并一直位于应答端。

关于“妈妈”套接字，正如我们小时候所经历的，只能等她向你开口时你们才能对话。妈妈不像爸爸那么开明，也不会像DEALER套接字一样接受模棱两可的回答。所以，想和REQ套接字对话只有等它主动发出请求后才行，之后它就会一直等待你的回答，不管有多久。

“爸爸”套接字则给人一种强硬、冷漠的感觉，他只做一件事：无论你提出什么问题，都会给出一个精确的回答。不要期望一个REP套接字会主动和你对话或是将你俩的交谈传达给别人，它不会这么做的。

我们通常认为请求-应答模式一定是有来有往、有去有回的过程，但实际上这个过程是可以异步进行的。我们只需获得相应节点的地址，即可通过ROUTER套接字来异步地发送消息。ROUTER是ZMQ中唯一一个可以定位消息来源的套接字。

我们对请求-应答模式下的路由做一个小结：

- 对于瞬时的套接字，ROUTER会动态生成一个UUID来标识它，因此从ROUTER中获取到的消息里会包含这个标识；
- 对于持久的套接字，可以自定义标识，ROUTER会如直接将该标识放入消息之中；
- 具有显式声明标识的节点可以连接到其他类型的套接字；
- 节点可以通过配置文件等机制提前获知对方节点的标识，作出相应的处理。

我们至少有三种模式来实现和ROUTER的连接：

- ROUTER-DEALER
- ROUTER-REQ
- ROUTER-REP

每种模式下我们都可以完全掌控消息的路由方式，但不同的模式会有不一样的应用场景和消息流，下一节开始我们会逐一解释。

自定义路由也有一些注意事项：

- 自定义路由让节点能够控制消息的去向，这一点有悖ØMQ的规则。使用自定义路由的唯一理由是ØMQ缺乏更多的路由算法供我们选择；
- 未来的ØMQ版本可能包含一些我们自定义的路由方式，这意味着我们现在设计的代码可能无法在新版本的ØMQ中运行，或者成为一种多余；
- 内置的路由机制是可扩展的，且对装置友好，但自定义路由就需要自己解决这些问题。

所以说自定义路由的成本是比较高的，更多情况下应当交由ØMQ来完成。不过既然我们已经讲到这儿了，就继续深入下去吧！

ROUTER-DEALER路由

ROUTER-DEALDER是一种最简单的路由方式。将ROUTER和多个DEALER相连接，用一种合适的算法来决定如何分发消息给DEALER。DEALER可以是一个黑洞（只负责处理消息，不给任何返回）、代理（将消息转发给其他节点）或是服务（会发送返回信息）。

如果你要求DEALER能够进行回复，那就要保证只有一个ROUTER连接到DEALER，因为DEALER并不知道哪个特定的节点在联系它，如果有多个节点，它会做负载均衡，将消息分发出去。但如果DEALER是一个黑洞，那就可以连接任何数量的节点。

ROUTER-DEALER路由可以用来做什么呢？如果DEALER会将它完成任务的时间回复给ROUTER，那ROUTER就可以知道这个DEALER的处理速度有多快了。因为ROUTER和DEALER都是异步的套接字，所以我们要用zmq_poll()来处理这种情况。

下面例子中的两个DEALER不会返回消息给ROUTER，我们的路由采用加权随机算法：发送两倍多的信息给其中的一个DEALER。

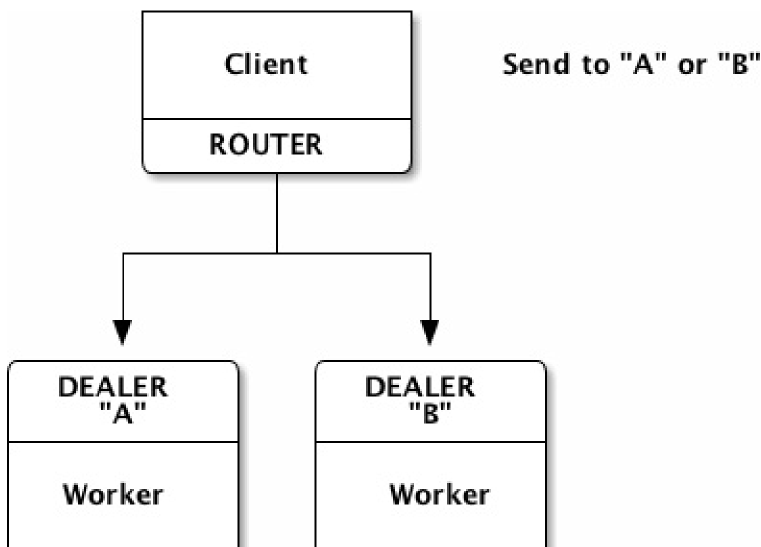


Figure 5 — Router to dealer custom routing

rtdealer.c

```
//
// 自定义ROUTER-DEALER路由
//
// 这个实例是单个进程，这样方便启动。
// 每个线程都有自己的ZMQ上下文，所以可以认为是多个进程在运行。
//
#include "zhelpers.h"
#include <pthread.h>
```

```
// 这里定义了两个worker，其代码是一样的。
//
static void *
worker_task_a (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 我们只接受到消息的第二部分
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("A received: %d\n", total);
            break;
        }
        total++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

static void *
worker_task_b (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "B", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 我们只接受到消息的第二部分
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("B received: %d\n", total);
            break;
        }
        total++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}
```

```
int main (void)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (client, "ipc://routing.ipc");

    pthread_t worker;
    pthread_create (&worker, NULL, worker_task_a, NULL);
    pthread_create (&worker, NULL, worker_task_b, NULL);

    // 等待线程连接至套接字，否则我们发送的消息将不能被正确路由
    sleep (1);

    // 发送10个任务，给A两倍多的量
    int task_nbr;
    srandom ((unsigned) time (NULL));
    for (task_nbr = 0; task_nbr < 10; task_nbr++) {
        // 发送消息的两个部分：第一部分是目标地址
        if (randof (3) > 0)
            s_sendmore (client, "A");
        else
            s_sendmore (client, "B");

        // 然后是任务
        s_send (client, "This is the workload");
    }
    s_sendmore (client, "A");
    s_send (client, "END");

    s_sendmore (client, "B");
    s_send (client, "END");

    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

对上述代码的两点说明：

- ROUTER并不知道DEALER何时会准备好，我们可以用信号机制来解决，但为了不让这个例子太过复杂，我们就用sleep(1)的方式来处理。如果没有这句话，那ROUTER一开始发出的消息将无法被路由，ØMQ会丢弃这些消息。
- 需要注意的是，除了ROUTER会丢弃无法路由的消息外，PUB套接字当没有SUB连接它时也会丢弃发送出去的消息。其他套接字则会将无法发送的消息存储起来，直到有节点来处理它们。

在将消息路由给DEALER时，我们手工建立了这样一个信封：

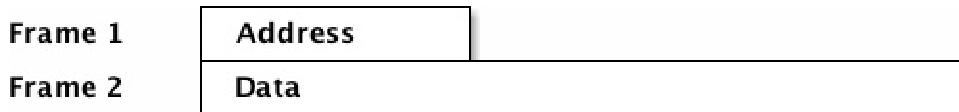


Figure 6 – Routing envelope for dealer

ROUTER套接字会移除第一帧，只将第二帧的内容传递给相应的DEALER。当DEALER发送消息给ROUTER时，只会发送一帧，ROUTER会在外层包裹一个信封（添加第一帧），返回给我们。

如果你定义了一个非法的信封地址，ROUTER会直接丢弃该消息，不作任何提示。对于这一点我们也无能为力，因为出现这种情况只有两种可能，一是要送达的目标节点不复存在了，或是程序中错误地指定了目标地址。如何才能知道消息会被正确地路由？唯一的方法是让路由目标发送一些反馈消息给我们。后面几章会讲述这一点。

DEALER的工作方式就像是PUSH和PULL的结合。但是，我们不能用PULL或PUSH去构建请求-应答模式。

最近最少使用算法路由（LRU模式）

我们之前讲过REQ套接字永远是对话的发起方，然后等待对方回答。这一特性可以让我们能够保持多个REQ套接字等待调配。换句话说，REQ套接字会告诉我们它已经准备好了。

你可以将ROUTER和多个REQ相连，请求-应答的过程如下：

- REQ发送消息给ROUTER
- ROUTER返回消息给REQ
- REQ发送消息给ROUTER
- ROUTER返回消息给REQ
- ...

和DEALER相同，REQ只能和一个ROUTER连接，除非你想做类似多路冗余路由这样的事（我甚至不想在这里解释），其复杂度会超过你的想象并迫使你放弃的。

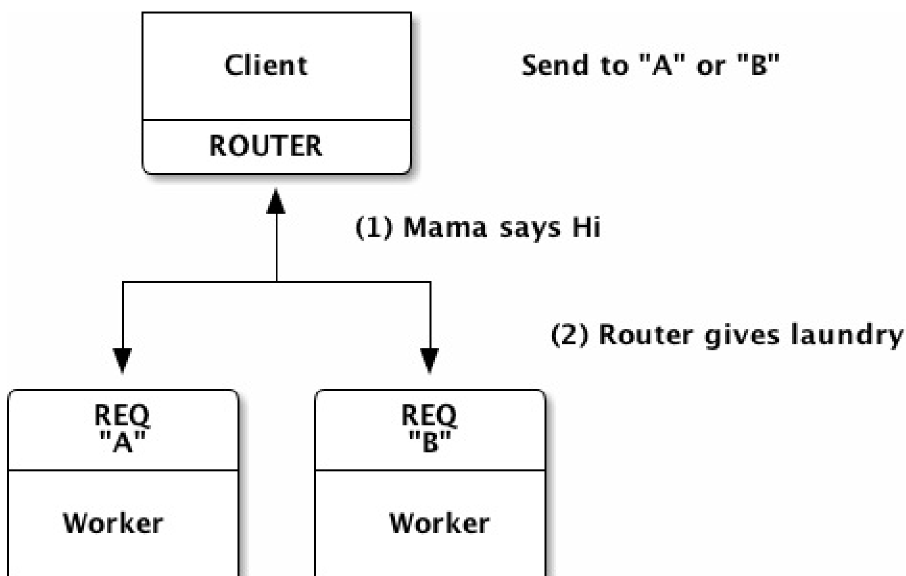


Figure 7 — Router to mama custom routing

ROUTER-REQ模式可以用来做什么？最常用的做法就是最近最少使用算法（LRU）路由了，ROUTER发出的请求会让等待最久的REQ来处理。请看示例：

```
//
// 自定义ROUTER-REQ路由
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task(void *args) {
    void *context = zmq_init(1);
    void *worker = zmq_socket(context, ZMQ_REQ);

    // s_set_id()函数会根据套接字生成一个可打印的字符串，
    // 并以此作为该套接字的标识。
    s_set_id(worker);
    zmq_connect(worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 告诉ROUTER我已经准备好了
        s_send(worker, "ready");

        // 从ROUTER中获取工作，直到收到结束的信息
        char *workload = s_recv(worker);
        int finished = (strcmp(workload, "END") == 0);
        free(workload);
        if (finished) {
            printf("Processed: %d tasks\n", total);
            break;
```

```
        }
        total++;

        // 随机等待一段时间
        s_sleep(randof(1000) + 1);
    }
    zmq_close(worker);
    zmq_term(context);
    return NULL;
}

int main(void) {
    void *context = zmq_init(1);
    void *client = zmq_socket(context, ZMQ_ROUTER);
    zmq_bind(client, "ipc://routing.ipc");
    srandom((unsigned) time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, NULL);
    }
    int task_nbr;
    for (task_nbr = 0; task_nbr < NBR_WORKERS * 10; task_nbr++) {
        // 最近最少使用的worker就在消息队列中
        char *address = s_recv(client);
        char *empty = s_recv(client);
        free(empty);
        char *ready = s_recv(client);
        free(ready);

        s_sendmore(client, address);
        s_sendmore(client, "");
        s_send(client, "This is the workload");
        free(address);
    }
    // 通知所有REQ套接字结束工作
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        char *address = s_recv(client);
        char *empty = s_recv(client);
        free(empty);
        char *ready = s_recv(client);
        free(ready);

        s_sendmore(client, address);
        s_sendmore(client, "");
        s_send(client, "END");
        free(address);
    }
    zmq_close(client);
    zmq_term(context);
    return 0;
}
```

在这个示例中，实现LRU算法并没有用到特别的数据结构，因为ØMQ的消息队列机制已经提供了等价的实现。一个更为实际的LRU算法应该将已准备好的worker收集起来，保存在一个队列中进行分配。以后我们会讲到这个例子。

程序的运行结果会将每个worker的执行次数打印出来。由于REQ套接字会随机等待一段时间，而我们也没有做负载均衡，所以我们希望看到的是每个worker执行相近的工作量。这也是程序执行的结果。

```
Processed: 8 tasks
Processed: 8 tasks
Processed: 11 tasks
Processed: 7 tasks
Processed: 9 tasks
Processed: 11 tasks
Processed: 14 tasks
Processed: 11 tasks
Processed: 11 tasks
Processed: 10 tasks
```

关于以上代码的几点说明：

- 我们不需要像前一个例子一样等待一段时间，因为REQ套接字会明确告诉ROUTER它已经准备好了。
- 我们使用了zhelpers.h提供的s_set_id()函数来为套接字生成一个可打印的字符串标识，这是为了让例子简单一些。在现实环境中，REQ套接字都是匿名的，你需要直接调用zmq_recv()和zmq_send()来处理消息，因为s_recv()和s_send()只能处理字符串标识的套接字。
- 更糟的是，我们使用了随机的标识，不要在现实环境中使用随机标识的持久套接字，这样做会将节点消耗殆尽。
- 如果你只是将上面的代码拷贝过来，没有充分理解，那你就像是看到蜘蛛人从屋顶上飞下来，你也照着做了，后果自负吧。

在将消息路由给REQ套接字时，需要注意一定的格式，即地址-空帧-消息：

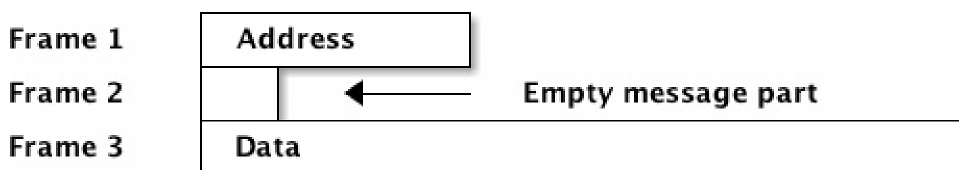


Figure 8 — Routing envelope for mama (REQ)

使用地址进行路由

在经典的请求-应答模式中，ROUTER一般不会和REP套接字通信，而是由DEALER去和REP通信。DEALER会将消息随机分发给多个REP，并获得结果。ROUTER更适合和REQ套接字通信。

我们应该记住，ØMQ的经典模型往往是运行得最好的，毕竟人走得多的路往往是条好路，如果不按常理出牌，那很有可能会跌入无敌深潭。下面我们就将ROUTER和REP进行连接，看看会发生什么。

REP套接字有两个特点：

- 它需要完成完整的请求-应答周期；
- 它可以接受任意大小的信封，并能完整地返回该信封。

在一般的请求-应答模式中，REP是匿名的，可以随时替换。因为我们这里在将自定义路由，就要做到将一条消息发送给REP A，而不是REP B。这样才能保证网络的一端是你，另一端是特定的REP。

ØMQ的核心理念之一是周边的节点应该尽可能的智能，且数量众多，而中间件则是固定和简单的。这就意味着周边节点可以向其他特定的节点发送消息，比如可以连接到一个特定的REP。这里我们先不讨论如何在多个节点之间进行路由，只看最后一步中ROUTER如何和特定的REP通信的。

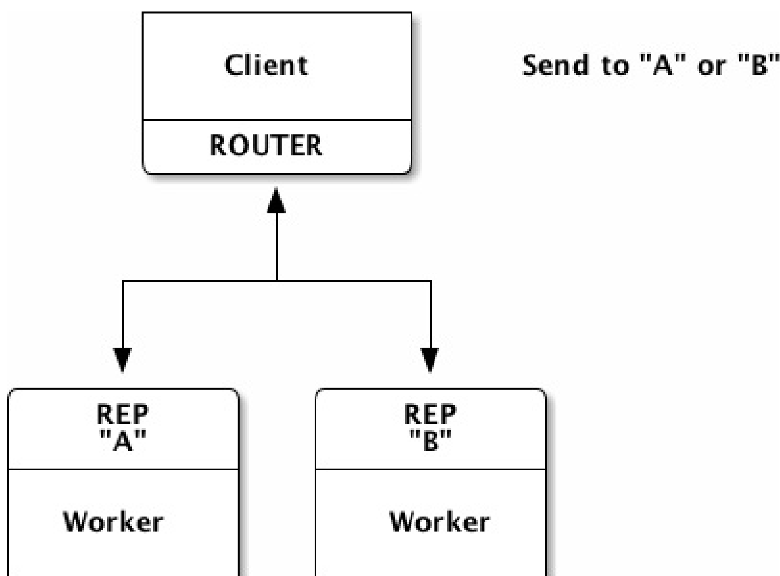


Figure 9 – Router to papa custom routing

这张图描述了以下事件：

- client有一条消息，将来会通过另一个ROUTER将该消息发送回去。这条信息包含了两个地址、一个空帧、以及消息内容；
- client将该条消息发送给了ROUTER，并指定了REP的地址；
- ROUTER将该地址移去，并以此决定其下哪个REP可以获得该消息；
- REP收到该条包含地址、空帧、以及内容的消息；
- REP将空帧之前的所有内容移去，交给worker去处理消息；
- worker处理完成后将回复交给REP；
- REP将之前保存好的信封包裹住该条回复，并发送给ROUTER；
- ROUTER在该条回复上又添加了一个注明REP的地址的帧。

这个过程看起来很复杂，但还是有必要取了解清楚的。只要记住，REP套接字会原封不动地将信封返回回去。

rtpapa.c

```
//
//  自定义ROUTER-REP路由
//
#include "zhelpers.h"

//  这里使用一个进程来强调事件发生的顺序性
int main (void)
{
    void *context = zmq_init (1);

    void *client = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (client, "ipc://routing.ipc");

    void *worker = zmq_socket (context, ZMQ_REP);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    //  等待worker连接
    sleep (1);

    //  发送REP的标识、地址、空帧、以及消息内容
    s_sendmore (client, "A");
    s_sendmore (client, "address 3");
    s_sendmore (client, "address 2");
    s_sendmore (client, "address 1");
    s_sendmore (client, "");
    s_send     (client, "This is the workload");

    //  worker只会得到消息内容
    s_dump (worker);

    //  worker不需要处理信封
    s_send (worker, "This is the reply");

    //  看看ROUTER里收到了什么
    s_dump (client);

    zmq_close (client);
    zmq_close (worker);
    zmq_term (context);
    return 0;
}
```

运行结果

```
----------------------------------------
[020] This is the workload
----------------------------------------
[001] A
[009] address 3
[009] address 2
[009] address 1
[000]
[017] This is the reply
```

关于以上代码的几点说明：

- 在现实环境中，ROUTER和REP套接字处于不同的节点。本例没有启用多进程，为的是让事件的发生顺序更为清楚。
- zmq_connect()并不是瞬间完成的，REP和ROUTER连接的时候是会花费一些时间的。在现实环境中，ROUTER无从得知REP是否已经连接成功了，除非得到REP的某些回应。本例中使用sleep(1)来处理这一问题，如果不这样做，那REP将无法获得消息（自己尝试一下吧）。
- 我们使用REP的套接字标识来进行路由，如果你不信，可以将消息发送给B，看看A能不能收到。
- 本例中的s_dump()等函数来自于zhelpers.h文件，可以看到在进行套接字连接时代码都是一样的，所以我们才能在ØMQ API的基础上搭建上层的API。等今后我们讨论到复杂应用程序的时候再详细说明。

要将消息路由给REP，我们需要创建它能辨别的信封：

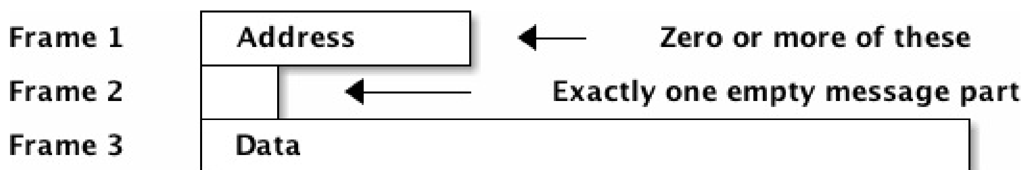


Figure 10 — Routing envelope for papa aka REP

请求-应答模式下的消息代理

这一节我们将对如何使用ØMQ消息信封做一个回顾，并尝试编写一个通用的消息代理装置。我们会建立一个队列装置来连接多个client和worker，装置的路由算法可以由我们自己决定。这里我们选择最近最少使用算法，因为这和负载均衡一样比较实用。

首先让我们回顾一下经典的请求-应答模型，尝试用它建立一个不断增长的巨型服务网络。最基本的请求-应答模型是：

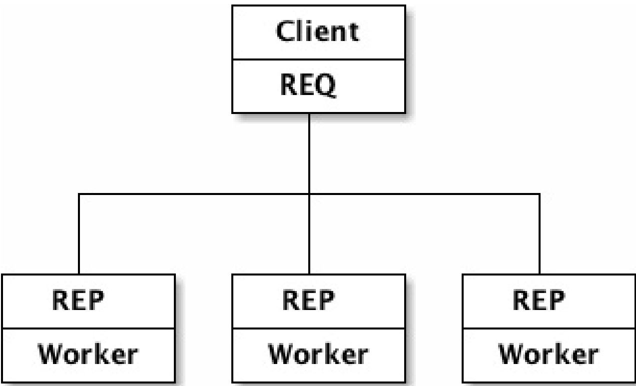


Figure 11 — Basic request reply

这个模型支持多个REP套接字，但如果我们想支持多个REQ套接字，就需要增加一个中间件，它通常是ROUTER和DEALER的结合体，简单将两个套接字之间的信息进行搬运，因此可以用现成的ZMQ_QUEUE装置来实现：

```
+--------+  +--------+  +--------+
| Client |  | Client |  | Client |
+--------+  +--------+  +--------+
|  REQ   |  |  REQ   |  |  REQ   |
+---+----+  +---+----+  +---+----+
    |           |           |
    +-----------+-----------+
                |
            +---+----+
            | ROUTER |
            +--------+
            | Device |
            +--------+
            | DEALER |
            +---+----+
                |
    +-----------+-----------+
    |           |           |
+---+----+  +---+----+  +---+----+
|  REP   |  |  REP   |  |  REP   |
+--------+  +--------+  +--------+
| Worker |  | Worker |  | Worker |
+--------+  +--------+  +--------+


Figure # - Stretched request-reply
```

这种结构的关键在于，ROUTER会将消息来自哪个REQ记录下来，生成一个信封。DEALER和REP套接字在传输消息的过程中不会丢弃或更改信封的内容，这样当消息返回给ROUTER时，它就知道应该发送给哪个REQ了。这个模型中的REP套接字是匿名的，并没有特定的地址，所以只能提供同一种服务。

上述结构中，对REP的路由我们使用了DEADER自带的负载均衡算法。但是，我们想用LRU算法来进行路由，这就要用到ROUTER-REP模式：

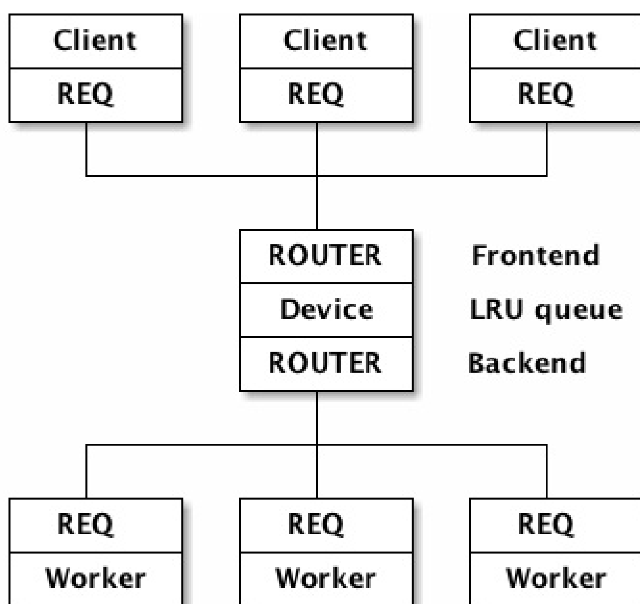


Figure 12 – Stretched request-reply with LRU

这个ROUTER-ROUTER的LRU队列不能简单地在两个套接字间搬运消息，以下代码会比较复杂，不过在请求-应答模式中复用性很高。

lruqueue.c

```
//
//  使用LRU算法的装置
//  client和worker处于不同的线程中
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  出队操作，使用一个可存储任何类型的数组实现
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) - sizeof (

//  使用REQ套接字实现基本的请求-应答模式
//  由于s_send()和s_recv()不能处理0MQ的二进制套接字标识，
//  所以这里会生成一个可打印的字符串标识。
//
static void *
client_task (void *args)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          //  设置可打印的标识
    zmq_connect (client, "ipc://frontend.ipc");
```

```
    //  发送请求并获取应答信息
    s_send (client, "HELLO");
    char *reply = s_recv (client);
    printf ("Client: %s\n", reply);
    free (reply);
    zmq_close (client);
    zmq_term (context);
    return NULL;
}

//  worker使用REQ套接字实现LRU算法
//
static void *
worker_task (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  设置可打印的标识
    zmq_connect (worker, "ipc://backend.ipc");

    //  告诉代理worker已经准备好
    s_send (worker, "READY");

    while (1) {
        //  将消息中空帧之前的所有内容（信封）保存起来，
        //  本例中空帧之前只有一帧，但可以有更多。
        char *address = s_recv (worker);
        char *empty = s_recv (worker);
        assert (*empty == 0);
        free (empty);

        //  获取请求，并发送回应
        char *request = s_recv (worker);
        printf ("Worker: %s\n", request);
        free (request);

        s_sendmore (worker, address);
        s_sendmore (worker, "");
        s_send     (worker, "OK");
        free (address);
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

int main (void)
{
    //  准备0MQ上下文和套接字
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (frontend, "ipc://frontend.ipc");
```

```
    zmq_bind (backend,  "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_task, NULL);
    }
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }
    //  LRU逻辑
    //  - 一直从backend中获取消息；当有超过一个worker空闲时才从frontend获
    //  - 当woker回应时，会将该worker标记为已准备好，并转发woker的回应给cl
    //  - 如果client发送了请求，就将该请求转发给下一个worker

    //  存放可用worker的队列
    int available_workers = 0;
    char *worker_queue [10];

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, available_workers? 2: 1, -1);

        //  处理backend中worker的队列
        if (items [0].revents & ZMQ_POLLIN) {
            //  将worker的地址入队
            char *worker_addr = s_recv (backend);
            assert (available_workers < NBR_WORKERS);
            worker_queue [available_workers++] = worker_addr;

            //  跳过空帧

            char *empty = s_recv (backend);
            assert (empty [0] == 0);
            free (empty);

            // 第三帧是“READY”或是一个client的地址
            char *client_addr = s_recv (backend);

            //  如果是一个应答消息，则转发给client
            if (strcmp (client_addr, "READY") != 0) {
                empty = s_recv (backend);
                assert (empty [0] == 0);
                free (empty);
                char *reply = s_recv (backend);
                s_sendmore (frontend, client_addr);
                s_sendmore (frontend, "");
                s_send     (frontend, reply);
```

```
                free (reply);
                if (--client_nbr == 0)
                    break;      //  处理N条消息后退出
            }
            free (client_addr);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取下一个client的请求，交给空闲的worker处理
            //  client请求的消息格式是：[client地址][空帧][请求内容]
            char *client_addr = s_recv (frontend);
            char *empty = s_recv (frontend);
            assert (empty [0] == 0);
            free (empty);
            char *request = s_recv (frontend);

            s_sendmore (backend, worker_queue [0]);
            s_sendmore (backend, "");
            s_sendmore (backend, client_addr);
            s_sendmore (backend, "");
            s_send     (backend, request);

            free (client_addr);
            free (request);

            //  将该worker的地址出队
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

这段程序有两个关键点：1、各个套接字是如何处理信封的；2、LRU算法。我们先来看信封的格式。

我们知道REQ套接字在发送消息时会向头部添加一个空帧，接收时又会自动移除。我们要做的就是在传输消息时满足REQ的要求，处理好空帧。另外还要注意，ROUTER会在所有收到的消息前添加消息来源的地址。

现在我们就将完整的请求-应答流程走一遍，我们将client套接字的标识设为“CLIENT”，worker的设为“WORKER”。以下是client发送的消息：

Frame 1	5	HELLO	Data part

Figure 13 – Message that client sends

代理从ROUTER中获取到的消息格式如下：

Frame	Size	Content	Description
Frame 1	6	CLIENT	Identity of client
Frame 2	0		Empty message part
Frame 3	5	HELLO	Data part

Figure 14 — Message coming in on frontend

代理会从LRU队列中获取一个空闲woker的地址，作为信封附加在消息之上，传送给ROUTER。注意要添加一个空帧。

Frame	Size	Content	Description
Frame 1	6	WORKER	Identity of worker
Frame 2	0		Empty message part
Frame 3	6	CLIENT	Identity of client
Frame 4	0		Empty message part
Frame 5	5	HELLO	Data part

Figure 15 — Message sent to backend

REQ（worker）收到消息时，会将信封和空帧移去：

Frame	Size	Content	Description
Frame 1	6	CLIENT	Identity of client
Frame 2	0		Empty message part
Frame 3	5	HELLO	Data part

Figure 16 — Message delivered to worker

可以看到，worker收到的消息和client端ROUTER收到的消息是一致的。worker需要将该消息中的信封保存起来，只对消息内容做操作。

在返回的过程中：

- worker通过REQ传输给device消息[client地址][空帧][应答内容]；
- device从worker端的ROUTER中获取到[worker地址][空帧][client地址][空帧][应答内容]；
- device将worker地址保存起来，并发送[client地址][空帧][应答内容]给client端的ROUTER；
- client从REQ中获得到[应答内容]。

然后再看看LRU算法，它要求client和worker都使用REQ套接字，并正确的存储和返回消息信封，具体如下：

- 创建一组poll，不断地从backend（worker端的ROUTER）获取消息；只有当有空闲的worker时才从frontend（client端的ROUTER）获取消息；
- 循环执行poll

- 如果backend有消息，只有两种情况：1）READY消息（该worker已准备好，等待分配）；2）应答消息（需要转发给client）。两种情况下我们都会保存worker的地址，放入LRU队列中，如果有应答内容，则转发给相应的client。
- 如果frontend有消息，我们从LRU队列中取出下一个worker，将该请求发送给它。这就需要发送[worker地址][空帧][client地址][空帧][请求内容]到worker端的ROUTER。

我们可以对该算法进行扩展，如在worker启动时做一个自我测试，计算出自身的处理速度，并随READY消息发送给代理，这样代理在分配工作时就可以做相应的安排。

ØMQ上层API的封装

使用ØMQ提供的API操作多段消息时是很麻烦的，如以下代码：

```
while (1) {
    //  将消息中空帧之前的所有内容（信封）保存起来，
    //  本例中空帧之前只有一帧，但可以有更多。
    char *address = s_recv (worker);
    char *empty = s_recv (worker);
    assert (*empty == 0);
    free (empty);

    //  获取请求，并发送回应
    char *request = s_recv (worker);
    printf ("Worker: %s\n", request);
    free (request);
    s_sendmore (worker, address);
    s_sendmore (worker, "");
    s_send     (worker, "OK");
    free (address);
}
```

这段代码不满足重用的需求，因为它只能处理一个帧的信封。事实上，以上代码已经做了一些封装了，如果调用ØMQ底层的API的话，代码就会更加冗长：

```
while (1) {
    //  将消息中空帧之前的所有内容（信封）保存起来，
    //  本例中空帧之前只有一帧，但可以有更多。
    zmq_msg_t address;
    zmq_msg_init (&address);
    zmq_recv (worker, &address, 0);

    zmq_msg_t empty;
    zmq_msg_init (&empty);
    zmq_recv (worker, &empty, 0);

    //  获取请求，并发送回应
    zmq_msg_t payload;
    zmq_msg_init (&payload);
    zmq_recv (worker, &payload, 0);

    int char_nbr;
    printf ("Worker: ");
    for (char_nbr = 0; char_nbr < zmq_msg_size (&payload); char_nbr
        printf ("%c", *(char *) (zmq_msg_data (&payload) + char_nbr
    printf ("\n");

    zmq_msg_init_size (&payload, 2);
    memcpy (zmq_msg_data (&payload), "OK", 2);

    zmq_send (worker, &address, ZMQ_SNDMORE);
    zmq_close (&address);
    zmq_send (worker, &empty, ZMQ_SNDMORE);
    zmq_close (&empty);
    zmq_send (worker, &payload, 0);
    zmq_close (&payload);
}
```

我们理想中的API是可以一步接收和处理完整的消息，包括信封。ØMQ底层的API并不是为此而涉及的，但我们可以在它上层做进一步的封装，这也是学习ØMQ的过程中很重要的内容。

想要编写这样一个API还是很有难度的，因为我们要避免过于频繁地复制数据。此外，ØMQ用“消息”来定义多段消息和多段消息中的一部分，同时，消息又可以是字符串消息或者二进制消息，这也给编写API增加的难度。

解决方法之一是使用新的命名方式：字符串（s_send()和s_recv()中已经在用了）、帧（消息的一部分）、消息（一个或多个帧）。以下是用新的API重写的worker：

```
while (1) {
    zmsg_t *zmsg = zmsg_recv (worker);
    zframe_print (zmsg_last (zmsg), "Worker: ");
    zframe_reset (zmsg_last (zmsg), "OK", 2);
    zmsg_send (&zmsg, worker);
}
```

用4行代码代替22行代码是个不错的选择，而且更容易读懂。我们可以用这种理念继续编写其他的API，希望可以实现以下功能：

- 自动处理套接字。每次都要手动关闭套接字是很麻烦的事，手动定义过期时间也不是太有必要，所以，如果能在关闭上下文时自动关闭套接字就太好了。
- 便捷的线程管理。基本上所有的ØMQ应用都会用到多线程，但POSIX的多线程接口用起来并不是太方便，所以也可以封装一下。
- 便捷的时钟管理。想要获取毫秒数、或是暂停运行几毫秒都不太方便，我们的API应该提供这个接口。
- 一个能够替代zmq_poll()的反应器。poll循环很简单，但比较笨拙，会造成重复代码：计算时间、处理套接字中的信息等。若有一个简单的反应器来处理套接字的读写以及时间的控制，将会很方便。
- 恰当地处理Ctrl-C按键。我么已经看到如何处理中断了，最好这一机制可以用到所有的程序里。

我们可以用czmq来实现以上的需求。这个扩展很早就有了，提供了很多ØMQ的上层封装，甚至是数据结构（哈希、链表等）。

以下是用czmq重写的LRU代理：

lruqueue2.c

```
//
//  LRU消息队列装置，使用czmq库实现
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  worker准备就绪的信息

//  使用REQ套接字实现基本的请求-应答模式
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");
```

```
    //  发送请求并接收应答
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，实现LRU路由
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  告知代理worker已准备就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  接收消息并处理
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  终止
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "ipc://frontend.ipc");
    zsocket_bind (backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
```

```
        zthread_new (ctx, worker_task, NULL);

    //  LRU逻辑
    //  - 一直从backend中获取消息；当有超过一个worker空闲时才从frontend获取
    //  - 当woker回应时，会将该worker标记为已准备好，并转发woker的回应给cl
    //  - 如果client发送了请求，就将该请求转发给下一个worker

    //  存放可用worker的队列
    zlist_t *workers = zlist_new ();

    while (1) {
        //  初始化poll
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当有可用的worker时，从frontend获取消息
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  中断

        //  对backend发来的消息进行处理
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker的地址进行LRU路由
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  如果不是READY消息，则转发给client
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取client发来的请求，转发给worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  如果完成了，则进行一些清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
```

```
    return 0;
}
```

czmq提供了一个简单的中断机制，当按下Ctrl-C时程序会终止ØMQ的运行，并返回-1，errno设置为EINTR。程序中断时，czmq的recv方法会返回NULL，所以你可以用下面的代码来作判断：

```
while (1) {
    zstr_send (client, "HELLO");
    char *reply = zstr_recv (client);
    if (!reply)
        break; // 中断
    printf ("Client: %s\n", reply);
    free (reply);
    sleep (1);
}
```

如果使用zmq_poll()函数，则可以这样判断：

```
int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
if (rc == -1)
    break; // 中断
```

上例中还是使用了原生的zmq_poll()方法，也可以使用czmq提供的zloop反应器来实现，它可以做到：

- 从任意套接字上获取消息，也就是说只要套接字有消息就可以触发函数；
- 停止读取套接字上的消息；
- 设置一个时钟，定时地读取消息。

zloop内部当然是使用zmq_poll()实现的，但它可以做到动态地增减套接字上的监听器，重构poll池，并根据poll的超时时间来计算下一个时钟触发事件。

使用这种反应器模式后，我们的代码就更简洁了：

```
zloop_t *reactor = zloop_new ();
zloop_reader (reactor, self->backend, s_handle_backend, self);
zloop_start (reactor);
zloop_destroy (&reactor);
```

对消息的实际处理放在了程序的其他部分，并不是所有人都会喜欢这种风格，但zloop的确是将定时器和套接字的行为融合在了一起。在以后的例子中，我们会用zmq_poll()来处理简单的示例，使用zloop来处理复杂的。

下面我们用zloop来重写LRU队列装置

lruqueue3.c

```
//
//  LRU队列装置，使用czmq及其反应器模式实现
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  woker已准备就绪的消息

//  使用REQ实现基本的请求-应答模式
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  发送请求并接收应答
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字来实现路由
//
static void *
worker_task (void *arg_ptr)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  告诉代理worker已经准备就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  获取消息并处理
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断
```

```
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  LRU队列处理器结构，将要传给反应器
typedef struct {
    void *frontend;             //  监听client
    void *backend;              //  监听worker
    zlist_t *workers;           //  可用的worker列表
} lruqueue_t;


//  处理frontend端的消息
int s_handle_frontend (zloop_t *loop, void *socket, void *arg)
{
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  如果没有可用的worker，则停止监听frontend
        if (zlist_size (self->workers) == 0)
            zloop_cancel (loop, self->frontend);
    }
    return 0;
}

//  处理backend端的消息
int s_handle_backend (zloop_t *loop, void *socket, void *arg)
{
    //  使用worker的地址进行LRU路由
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *address = zmsg_unwrap (msg);
        zlist_append (self->workers, address);

        //  当有可用worker时增加frontend端的监听
        if (zlist_size (self->workers) == 1)
            zloop_reader (loop, self->frontend, s_handle_frontend,

        //  如果是worker发送来的应答，则转发给client
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
```

```
    return 0;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    lruqueue_t *self = (lruqueue_t *) zmalloc (sizeof (lruqueue_t)
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "ipc://frontend.ipc");
    zsocket_bind (self->backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  可用worker的列表
    self->workers = zlist_new ();

    //  准备并启动反应器
    zloop_t *reactor = zloop_new ();
    zloop_reader (reactor, self->backend, s_handle_backend, self);
    zloop_start (reactor);
    zloop_destroy (&reactor);

    //  结束之后的清理工作
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

要正确处理Ctrl-C还是有点困难的，如果你使用zctx类，那它会自动进行处理，不过也需要代码的配合。若zmq_poll()返回了-1，或者recv方法（zstr_recv, zframe_recv, zmsg_recv）返回了NULL，就必须退出所有的循环。另外，在最外层循环中增加!zctx_interrupted的判断也很有用。

异步C/S结构

在之前的ROUTER-DEALER模型中，我们看到了client是如何异步地和多个worker进行通信的。我们可以将这个结构倒置过来，实现多个client异步地和单个server进行通信：

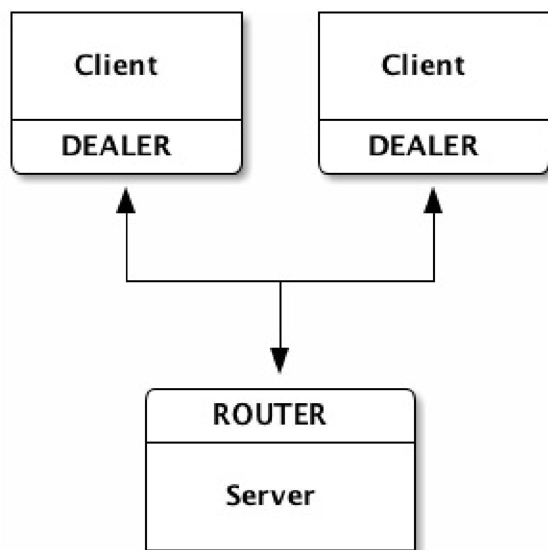


Figure 17 – Asynchronous Client Server

- client连接至server并发送请求；
- 每一次收到请求，server会发送0至N个应答；
- client可以同时发送多个请求而不需要等待应答；
- server可以同时发送多个应答二不需要新的请求。

asyncsrd.c

```
//
//  异步C/S模型（DEALER-ROUTER）
//

#include "czmq.h"

//  ---------------------------------------------------------------
//  这是client端任务，它会连接至server，每秒发送一次请求，同时收集和打印应答；
//  我们会运行多个client端任务，使用随机的标识。

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    //  设置随机标识，方便跟踪
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10(
    zsockopt_set_identity (client, identity);
    zsocket_connect (client, "tcp://localhost:5570");

    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    int request_nbr = 0;
    while (1) {
        //  从poll中获取消息，每秒一次
        int centitick;
```

```
        for (centitick = 0; centitick < 100; centitick++) {
            zmq_poll (items, 1, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  ---------------------------------------------------------------------
//  这是server端任务，它使用多线程机制将请求分发给多个worker，并正确返回应答
//  一个worker只能处理一次请求，但client可以同时发送多个请求。

static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    zctx_t *ctx = zctx_new ();

    //  frontend套接字使用TCP和client通信
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");

    //  backend套接字使用inproc和worker通信
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    //  启动一个worker线程池，数量任意
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    //  使用队列装置连接backend和frontend，我们本来可以这样做：
    //      zmq_device (ZMQ_QUEUE, frontend, backend);
    //  但这里我们会自己完成这个任务，这样可以方便调试。

    //  在frontend和backend间搬运消息
    while (1) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            //puts ("Request from client:");
            //zmsg_dump (msg);
            zmsg_send (&msg, backend);
```

```
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            //puts ("Reply from worker:");
            //zmsg_dump (msg);
            zmsg_send (&msg, frontend);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  接收一个请求，随机返回多条相同的文字，并在应答之间做随机的延迟。
//
static void
server_worker (void *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (1) {
        //  DEALER套接字将信封和消息内容一起返回给我们
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *address = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
        zmsg_destroy (&msg);

        //  随机返回0至4条应答
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  暂停一段时间
            zclock_sleep (randof (1000) + 1);
            zframe_send (&address, worker, ZFRAME_REUSE + ZFRAME_MO
            zframe_send (&content, worker, ZFRAME_REUSE);
        }
        zframe_destroy (&address);
        zframe_destroy (&content);
    }
}


//  主程序用来启动多个client和一个server
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, server_task, NULL);

    //  运行5秒后退出
```

```
    zclock_sleep (5 * 1000);
    zctx_destroy (&ctx);
    return 0;
}
```

运行上面的代码，可以看到三个客户端有各自的随机标识，每次请求会获得零到多条回复。

- client每秒会发送一次请求，并获得零到多条应答。这要通过zmq_poll()来实现，但我们不能只每秒poll一次，这样将不能及时处理应答。程序中我们每秒取100次，这样一来server端也可以以此作为一种心跳（heartbeat），用来检测client是否还在线。
- server使用了一个worker池，每一个worker同步处理一条请求。我们可以使用内置的队列来搬运消息，但为了方便调试，在程序中我们自己实现了这一过程。你可以将注释的几行去掉，看看输出结果。

这段代码的整体架构如下图所示：

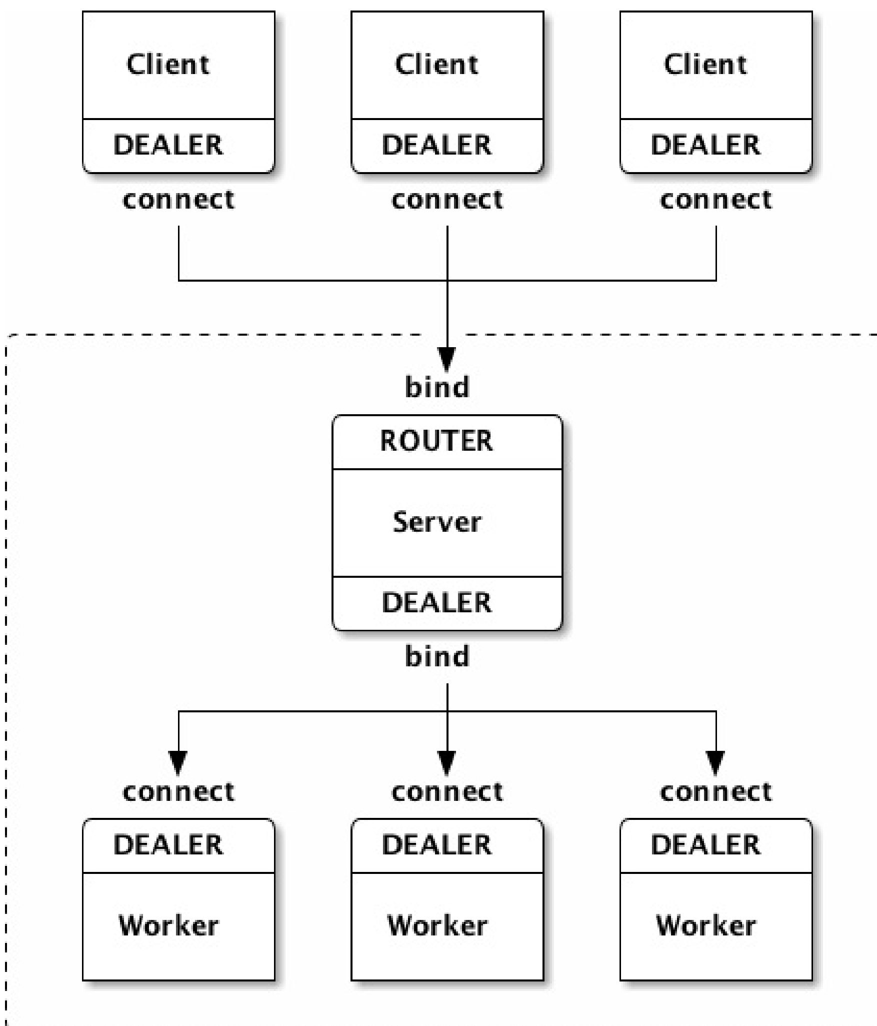


Figure 18 — Detail of asynchronous server

可以看到，client和server之间的连接我们使用的是DEALER-ROUTER，而server和worker的连接则用了DEALER-DEALER。如果worker是一个同步的线程，我们可以用REP。但是本例中worker需要能够发送多个应答，所以就需要使用DEALER这样的异步套接字。这里我们不需要对应答进行路由，因为所有的worker都是连接到一个server上的。

让我们看看路由用的信封，client发送了一条信息，server获取的信息中包含了client的地址，这样一来我们有两种可行的server-worker通信方案：

- worker收到未经标识的信息。我们使用显式声明的标识，配合ROUTER套接字来连接worker和server。这种设计需要worker提前告知ROUTER它的存在，这种LRU算法正是我们之前所讲述的。
- worker收到含有标识的信息，并返回含有标识的应答。这就要求worker能够处理好信封。

第二种涉及较为简单：

```
     client          server       frontend       worker
   [ DEALER ]<---->[ ROUTER <----> DEALER <----> DEALER ]
             1 part         2 parts       2 parts
```

当我们需要在client和server之间维持一个对话时，就会碰到一个经典的问题：client是不固定的，如果给每个client都保存一些消息，那系统资源很快就会耗尽。即使是和同一个client保持连接，因为使用的是瞬时的套接字（没有显式声明标识），那每次连接也相当于是一个新的连接。

想要在异步的请求中保存好client的信息，有以下几点需要注意：

- client需要发送心跳给server。本例中client每秒都会发送一个请求给server，这就是一种很可靠的心跳机制。
- 使用client的套接字标识来存储信息，这对瞬时和持久的套接字都有效；
- 检测停止心跳的client，如两秒内没有收到某个client的心跳，就将保存的状态丢弃。

实战：跨代理路由

让我们把目前所学到的知识综合起来，应用到实战中去。我们的大客户今天打来一个紧急电话，说是要构建一个大型的云计算设施。它要求这个云架构可以跨越多个数据中心，每个数据中心包含一组client和worker，且能共同协作。

我们坚信实践高于理论，所以就提议使用ZMQ搭建这样一个系统。我们的客户同意了，可能是因为他的确也想降低开发的成本，或是在推特上看到了太多ZMQ的好处。

细节详述

喝完几杯特浓咖啡，我们准备着手干了，但脑中有个理智的声音提醒我们应该在事前将问题分析清楚，然后再开始思考解决方案。云到底要做什么？我们如是问，客户这样回答：

- worker在不同的硬件上运作，但可以处理所有类型的任务。每个集群都有成百个worker，再乘以集群的个数，因此数量众多。
- client向worker指派任务，每个任务都是独立的，每个client都希望能找到对应的worker来处理任务，越快越好。client是不固定的，来去频繁。
- 真正的难点在于，这个架构需要能够自如地添加和删除集群，附带着集群中的client和worker。
- 如果集群中没有可用的worker，它便会将任务分派给其他集群中可以用的worker。
- client每次发送一个请求，并等待应答。如果X秒后他们没有获得应答，他们会重新发送请求。这一点我们不需要多做考虑，client端的API已经写好了。
- worker每次处理一个请求，他们的行为非常简单。如果worker崩溃了，会有另外的脚本启动他们。

听了以上的回答，我们又进一步追问：

- 集群之间会有一个更上层的网络来连接他们对吗？客户说是的。
- 我们需要处理多大的吞吐量？客户说，每个集群约有一千个client，单个client每秒会发送10次请求。请求包含的内容很少，应答也很少，每个不超过1KB。

我们进行了简单的计算，2500个client x 10次/秒 x 1000字节 x 双向 = 50MB/秒，或400Mb/秒，这对1Gb网络来说不成问题，可以使用TCP协议。

这样需求就很清晰了，不需要额外的硬件或协议来完成这件事，只要提供一个高效的路由算法，设计得缜密一些。我们首先从一个集群（数据中心）开始，然后思考如何来连接他们。

单个集群的架构

worker和client是同步的，我们使用LRU算法来给worker分配任务。每个worker都是等价的，所以我们不需要考虑服务的问题。worker是匿名的，client不会和某个特定的worker进行通信，因而我们不需要保证消息的送达以及失败后的重试等。

鉴于上文提过的原因，client和worker是不会直接通信的，这样一来就无法动态地添加和删除节点了。所以，我们的基础模型会使用一个请求-应答模式中使用过的代理结构。

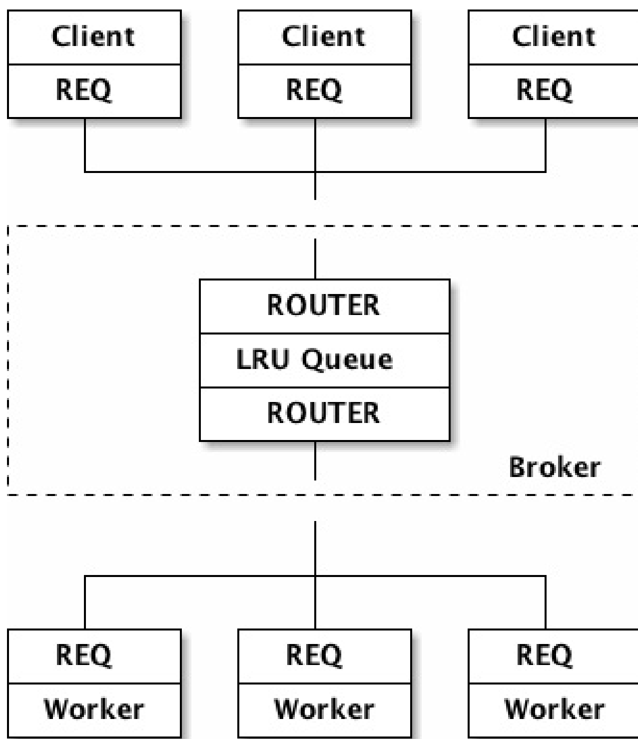


Figure 19 — Cluster architecture

多个集群的架构

下面我们将集群扩充到多个，每个集群有自己的一组client和worker，并使用代理相连接：

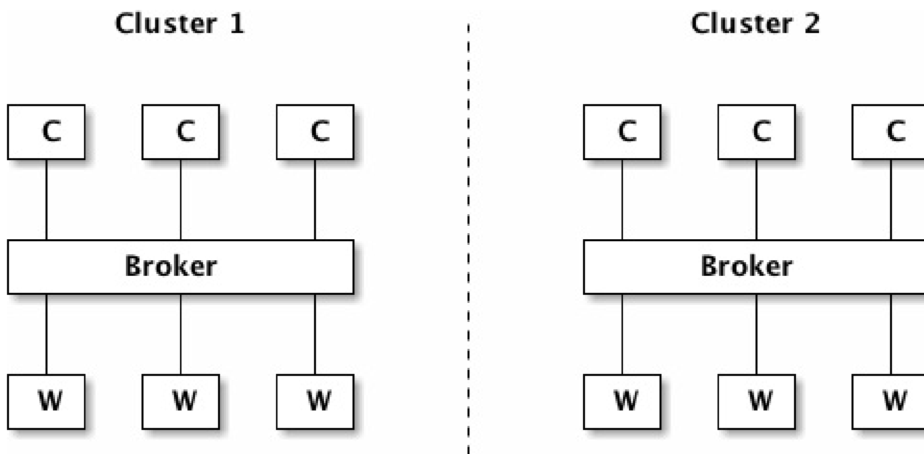


Figure 20 — Multiple clusters

问题在于：我们如何让一个集群的client和另一个集群的worker进行通信呢？有这样几种解决方案，我们来看看他们的优劣：

- client直接和多个代理相连接。优点在于我们可以不对代理和worker做改动，但client会变得复杂，并需要知悉整个架构的情况。如果我们想要添加第三或第四个集群，所有的client都会需要修改。我们相当于是将路由和容错功能写进client了，这并不是个好主意。

- worker直接和多个代理相连接。可是REQ类型的worker不能做到这一点，它只能应答给某一个代理。如果改用REP套接字，这样就不能使用LRU算法的队列代理了。这点肯定不行，在我们的结构中必须用LRU算法来管理worker。还有个方法是使用ROUTER套接字，让我们暂且称之为方案1。

- 代理之间可以互相连接，这看上去不错，因为不需要增加过多的额外连接。虽然我们不能随意地添加代理，但这个问题可以暂不考虑。这种情况下，集群中的worker和client不必理会整体架构，当代理有剩余的工作能力时便会和其他代理通信。这是方案2。

我们首先看看方案1，worker同时和多个代理进行通信：

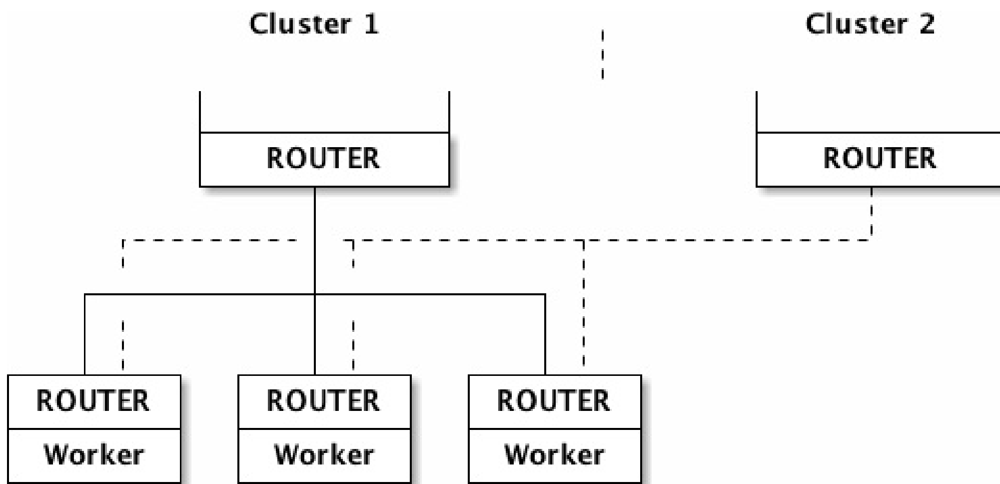


Figure 21 — Idea 1 — cross connected workers

这看上去很灵活，但却没有提供我们所需要的特性：client只有当集群中的worker不可用时才会去请求异地的worker。此外，worker的“已就绪”信号会同时发送给两个代理，这样就有可能同时获得两份任务。这个方案的失败还有一个原因：我们又将路由逻辑放在了边缘地带。

那来看看方案2，我们为各个代理建立连接，不修改worker和client：

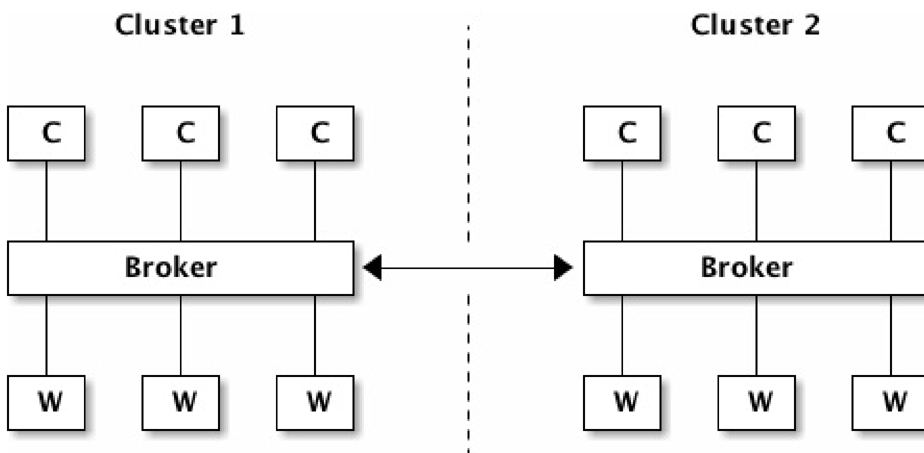


Figure 22 — Idea 2 — brokers talking to each other

这种设计的优势在于，我们只需要在一个地方解决问题就可以了，其他地方不需要修改。这就好像代理之间会秘密通信：伙计，我这儿有一些剩余的劳动力，如果你那儿忙不过来就跟我说，价钱好商量。

事实上，我们只不过是需要设计一种更为复杂的路由算法罢了：代理成为了其他代理的分包商。这种设计还有其他一些好处：

- 在普通情况下（如只存在一个集群），这种设计的处理方式和原来没有区别，当有多个集群时再进行其他动作。
- 对于不同的工作我们可以使用不同的消息流模式，如使用不同的网络链接。
- 架构的扩充看起来也比较容易，如有必要我们还可以添加一个超级代理来完成调度工作。

现在我们就开始编写代码。我们会将完整的集群写入一个进程，这样便于演示，而且稍作修改就能投入实际使用。这也是ZMQ的优美之处，你可以使用最小的开发模块来进行实验，最后方便地迁移到实际工程中。线程变成进程，消息模式和逻辑不需要改变。我们每个“集群”进程都包含client线程、worker线程、以及代理线程。

我们对基础模型应该已经很熟悉了：

- client线程使用REQ套接字，将请求发送给代理线程（ROUTER套接字）；
- worker线程使用REQ套接字，处理并应答从代理线程（ROUTER套接字）收到的请求；
- 代理会使用LRU队列和路由机制来管理请求。

联邦模式和同伴模式

连接代理的方式有很多，我们需要斟酌一番。我们需要的功能是告诉其他代理“我这里还有空闲的worker”，然后开始接收并处理一些任务；我们还需要能够告诉其他代理“够了够了，我这边的工作量也满了”。这个过程不一定要十分完美，有时我们确实会接收超过承受能力的工作量，但仍能逐步地完成。

最简单的方式称为联邦，即代理充当其他代理的client和worker。我们可以将代理的前端套接字连接至其他代理的后端套接字，反之亦然。提示一下，ZMQ中是可以将一个套接字绑定到一个端点，同时又连接至另一个端点的。

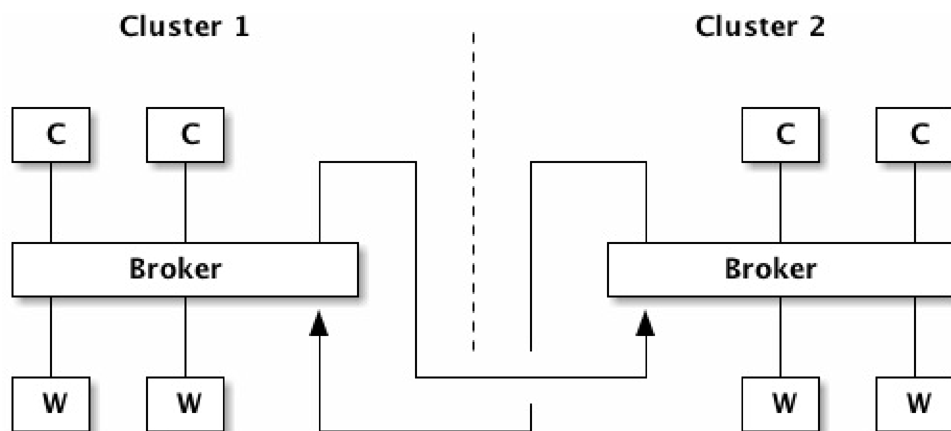


Figure 23 – Cross connected brokers in federation model

这种架构的逻辑会比较简单：当代理没有client时，它会告诉其他代理自己准备好了，并接收一个任务进行处理。但问题在于这种机制太简单了，联邦模式下的代理一次只能处理一个请求。如果client和worker是严格同步的，那么代理中的其他空闲worker将分配不到任务。我们需要的代理应该具备完全异步的特性。

但是，联邦模式对某些应用来说是非常好的，比如面向服务架构（SOA）。所以，先不要急着否定联邦模式，它只是不适用于LRU算法和集群负载均衡而已。

我们还有一种方式来连接代理：同伴模式。代理之间知道彼此的存在，并使用一个特殊的信道进行通信。我们逐步进行分析，假设有N个代理需要连接，每个代理则有N-1个同伴，所有代理都使用相同格式的消息进行通信。关于消息在代理之间的流通有两点需要注意：

- 每个代理需要告知所有同伴自己有多少空闲的worker，这是一则简单的消息，只是一个不断更新的数字，很显然我们会使用PUB-SUB套接字。这样一来，每个代理都会打开一个PUB套接字，不断告知外界自身的信息；同时又会打开一个SUB套接字，获取其他代理的信息。

- 每个代理需要以某种方式将工作任务交给其他代理，并能获取应答，这个过程需要是异步的。我们会使用ROUTER-ROUTER套接字来实现，没有其他选择。每个代理会使用两个这样的ROUTER套接字，一个用于接收任务，另一个用于分发任务。如果不使用两个套接字，那就需要额外的逻辑来判别收到的是请求还是应答，这就需要在消息中加入更多的信息。

另外还需要考虑的是代理和本地client和worker之间的通信。

The Naming Ceremony

代理中有三个消息流，每个消息流使用两个套接字，因此一共需要使用六个套接字。为这些套接字取一组好名字很重要，这样我们就不会在来回切换的时候找不着北。套接字是有一定任务的，他们的所完成的工作可以是命名的一部分。这样，当我们日后再重新阅读这些代码时，就不会显得太过陌生了。

以下是我们使用的三个消息流：

- 本地（local）的请求-应答消息流，实现代理和client、代理和worker之间的通信；
- 云端（cloud）的请求-应答消息流，实现代理和其同伴的通信；
- 状态（state）流，由代理和其同伴互相传递。

能够找到一些有意义的、且长度相同的名字，会让我们的代码对得比较整齐。可能他们并没有太多关联，但久了自然会习惯。

每个消息流会有两个套接字，我们之前一直称为“前端（frontend）”和“后端（backend）”。这两个名字我们已经使用很多次了：前端会负责接受信息或任务；后端会发送信息或任务给同伴。从概念上说，消息流都是从前往后的，应答则是从后往前。

因此，我们决定使用以下的命名方式：

- localfe / localbe

- cloudfe / cloudbe
- statefe / statebe

通信协议方面，我们全部使用ipc。使用这种协议的好处是，它能像tcp协议那样作为一种脱机通信协议来工作，而又不需要使用IP地址或DNS服务。对于ipc协议的端点，我们会命名为xxx-localfe/be、xxx-cloud、xxx-state，其中xxx代表集群的名称。

也许你会觉得这种命名方式太长了，还不如简单的叫s1、s2、s3……事实上，你的大脑并不是机器，阅读代码的时候不能立刻反应出变量的含义。而用上面这种“三个消息流，两个方向”的方式记忆，要比纯粹记忆“六个不同的套接字”来的方便。

以下是代理程序的套接字分布图：

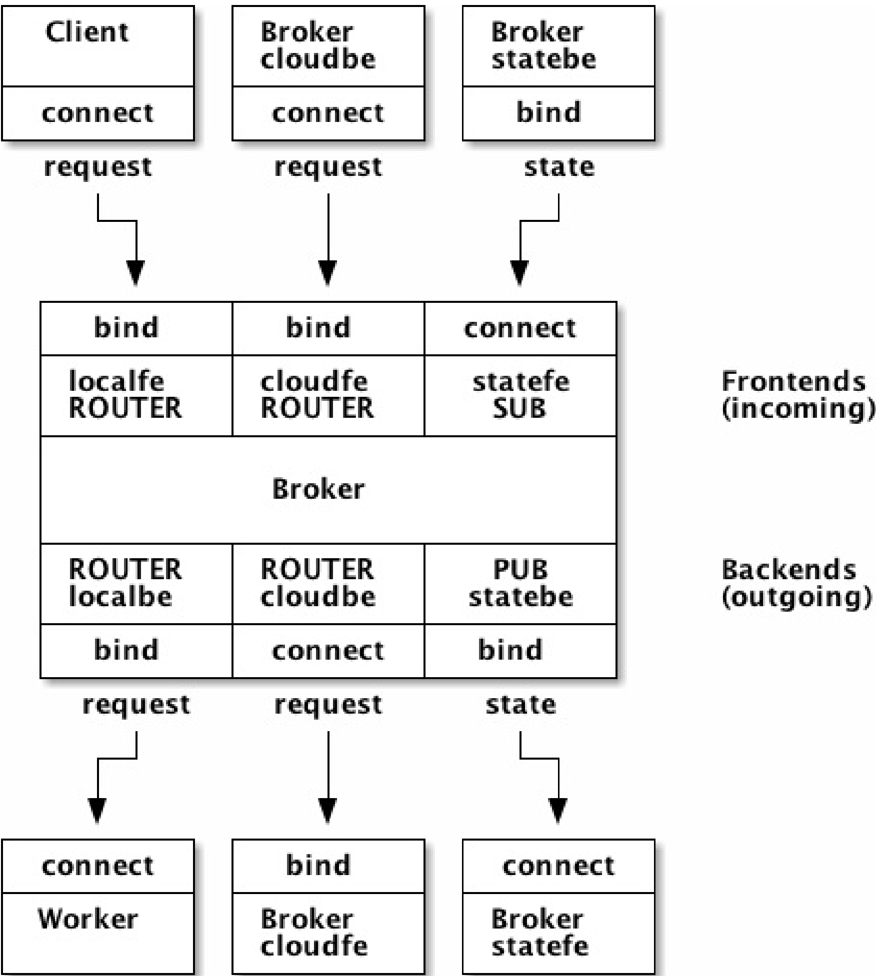


Figure 24 — Broker socket arrangement

请注意，我们会将cloudbe连接至其他代理的cloudfe，也会将statebe连接至其他代理的statefe。

状态流原型

由于每个消息流都有其巧妙之处，所以我们不会直接把所有的代码都写出来，而是分段编写和测试。当每个消息流都能正常工作了，我们再将其拼装成一个完整的应用程序。我们首先从状态流开始：

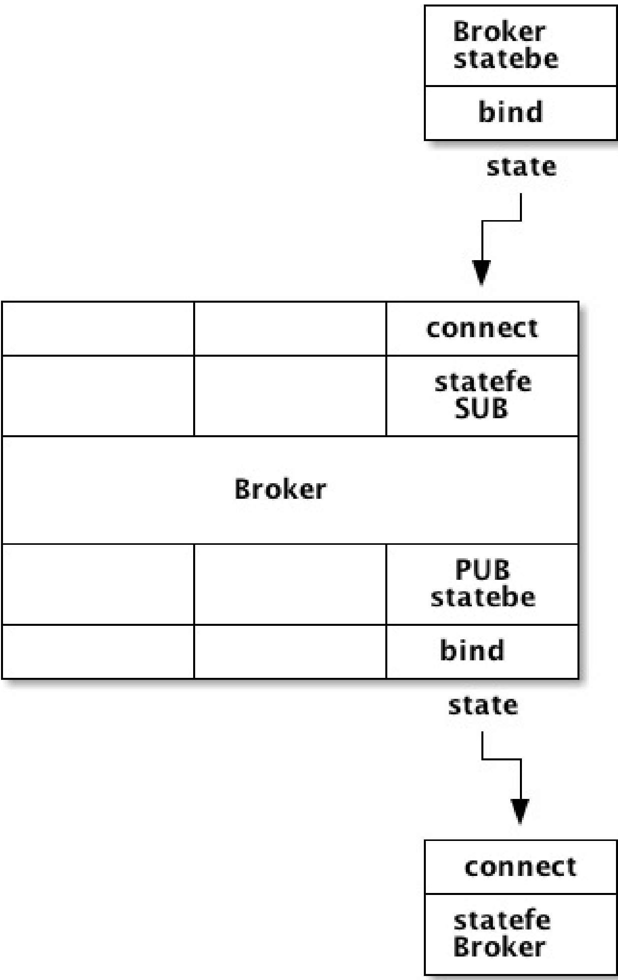


Figure 25 — The state flow

代码如下：

peering1: Prototype state flow in C

```
//
//  代理同伴模拟（第一部分）
//  状态流原型
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是各个同伴的名称
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    char *self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));
```

```
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  连接statefe套接字至所有同伴
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的状态流后端\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  发送并接受状态消息
    //  zmq_poll()函数使用的超时时间即心跳时间
    //
    while (1) {
        //  初始化poll对象列表
        zmq_pollitem_t items [] = {
            { statefe, 0, ZMQ_POLLIN, 0 }
        };
        //  轮询套接字活动，超时时间为1秒
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  处理接收到的状态消息
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("同伴代理 %s 有 %s 个worker空闲\n", peer_name, av
            free (peer_name);
            free (available);
        }
        else {
            //  发送随机数表示空闲的worker数
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

几点说明：

- 每个代理都需要有各自的标识，用以生成相应的ipc端点名称。真实环境中，代理需要使用TCP协议连接，这就需要一个更为完备的配置机制，我们会在以后的章节中谈到。

- 程序的核心是一个zmq_poll()循环，它会处理接收到消息，并发送自身的状态。只有当zmq_poll()因无法获得同伴消息而超时时我们才会发送自身状态，如果我们每次收到消息都去发送自身状态，那消息就会过量了。

- 发送的状态消息包含两帧，第一帧是代理自身的地址，第二帧是空闲的worker数。我们必须要告知同伴代理自身的地址，这样才能接收到请求，唯一的方法就是在消息中显示注明。

- 我们没有在SUB套接字上设置标识，否则就会在连接到同伴代理时获得过期的状态信息。

- 我们没有在PUB套接字上设置阈值（HWM），因为订阅者是瞬时的。我们也可以将阈值设置为1，但其实是没有必要的。

让我们编译这段程序，用它模拟三个集群，DC1、DC2、DC3。我们在不同的窗口中运行以下命令：

```
peering1 DC1 DC2 DC3  #  Start DC1 and connect to DC2 and DC3
peering1 DC2 DC1 DC3  #  Start DC2 and connect to DC1 and DC3
peering1 DC3 DC1 DC2  #  Start DC3 and connect to DC1 and DC2
```

每个集群都会报告同伴代理的状态，之后每隔一秒都会打印出自己的状态。

在现实编程中，我们不会通过定时的方式来发送自身状态，而是在状态发生改变时就发送。这看起来会很占用带宽，但其实状态消息的内容很少，而且集群间的连接是非常快速的。

如果我们想要以较为精确的周期来发送状态信息，可以新建一个线程，将statebe套接字打开，然后由主线程将不规则的状态信息发送给子线程，再由子线程定时发布这些消息。不过这种机制就需要额外的编程了。

本地流和云端流原型

下面让我们建立本地流和云端流的原型。这段代码会从client获取请求，并随机地分派给集群内的worker或其他集群。

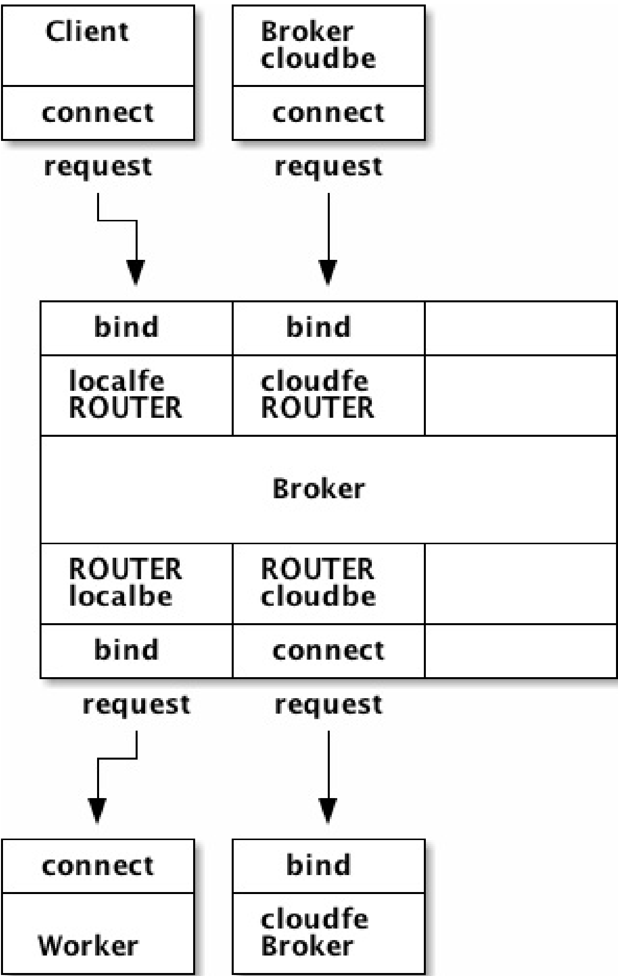


Figure 26 — The flow of tasks

在编写代码之前，让我们先描绘一下核心的路由逻辑，整理出一份简单而健壮的设计。

我们需要两个队列，一个队列用于存放从本地集群client收到的请求，另一个存放其他集群发送来的请求。一种方法是从本地和云端的前端套接字中获取消息，分别存入两个队列。但是这么做似乎是没有必要的，因为ZMQ套接字本身就是队列。所以，我们直接使用ZMQ套接字提供的缓存来作为队列使用。

这项技术我们在LRU队列装置中使用过，且工作得很好。做法是，当代理下有空闲的worker或能接收请求的其他集群时，才从套接字中获取请求。我们可以不断地从后端获取应答，然后路由回去。如果后端没有任何响应，那也就没有必要去接收前端的请求了。

所以，我们的主循环会做以下几件事：

- 轮询后端套接字，会从worker处获得"已就绪"的消息或是一个应答。如果是应答消息，则将其路由回集群client，或是其他集群。
- worker应答后即可标记为可用，放入队列并计数；
- 如果有可用的worker，就获取一个请求，该请求可能来自集群内的client，也可能是其他集群。随后将请求转发给集群内的worker，或是随机转发给其他集群。

这里我们只是随机地将请求发送给其他集群，而不是在代理中模拟出一个worker，进行集群间的任务分发。这看起来挺愚蠢的，不过目前尚可使用。

我们使用代理的标识来进行代理之前的消息路由。每个代理都有自己的名字，是在命令行中指定的。只要这些指定的名字和ZMQ为client自动生成的UUID不重复，那么我们就可以知道应答是要返回给client，还是返回给另一个集群。

下面是代码，有趣的部分已在程序中标注：

peering2: Prototype local and cloud flow in C

```
//
//  代理同伴模拟（第二部分）
//  请求-应答消息流原型
//
//  示例程序使用了一个进程，这样可以让程序变得简单，
//  每个线程都有自己的上下文对象，所以可以认为他们是多个进程。
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  消息：worker已就绪

//  代理名称；现实中，这个名称应该由某种配置完成
static char *self;

//  请求-应答客户端使用REQ套接字
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);

    while (1) {
        //  发送请求，接收应答
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  中断
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，并进行LRU路由
//
```

```
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告知代理worker已就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  处理消息
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断

        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}


int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是同伴代理的名称
    //
    if (argc < 2) {
        printf ("syntax: peering2 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  将cloudfe绑定至端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  将cloudbe连接至同伴代理的端点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
```

```
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的cloudfe端点\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  让用户告诉我们何时开始
    printf ("请确认所有代理已经启动，按任意键继续: ");
    getchar ();

    //  启动本地worker
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  启动本地client
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);

    //  有趣的部分
    //  -------------------------------------------------------------
    //  请求-应答消息流
    //  - 若本地有可用worker，则轮询获取本地或云端的请求；
    //  - 将请求路由给本地worker或其他集群。

    //  可用worker队列
    int capacity = 0;
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  如果没有可用worker，则继续等待
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  中断

        //  处理本地worker的应答
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);
```

```
            capacity++;

            //  如果是“已就绪”的信号，则不再进行路由
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  处理来自同伴代理的应答
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  中断
            //  我们不需要使用同伴代理的地址
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
        }
        //  如果应答消息中的地址是同伴代理的，则发送给它
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  将应答路由给本地client
        if (msg)
            zmsg_send (&msg, localfe);

        //  开始处理客户端请求
        //
        while (capacity) {
            zmq_pollitem_t frontends [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            rc = zmq_poll (frontends, 2, 0);
            assert (rc >= 0);
            int reroutable = 0;
            //  优先处理同伴代理的请求，避免资源耗尽
            if (frontends [1].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (cloudfe);
                reroutable = 0;
            }
            else
            if (frontends [0].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (localfe);
                reroutable = 1;
            }
            else
                break;      //  没有请求

            //  将20%的请求发送给其他集群
```

```
            //
            if (reroutable && argc > 2 && randof (5) == 0) {
                //  随地地路由给同伴代理
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv
                zmsg_send (&msg, cloudbe);
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  程序结束后的清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

在两个窗口中运行以上代码：

```
peering2 me you
peering2 you me
```

几点说明：

- zmsg类库让程序变得简单多了，这类程序显然应该成为我们ZMQ程序员必备的工具； 由于我们没有在程序中实现获取同伴代理状态的功能，所以先暂且认为他们都是有空闲worker的。现实中，我们不会将请求发送个一个不存在的同伴代理。
- 你可以让这段程序长时间地运行下去，看看会不会出现路由错误的消息，因为一旦错误，client就会阻塞。你可以试着将一个代理关闭，就能看到代理无法将请求路由给云端中的其他代理，client逐个阻塞，程序也停止打印跟踪信息。

组装

让我们将所有这些放到一段代码里。和之前一样，我们会在一个进程中完成所有工作。我们会将上文中的两个示例程序结合起来，编写出一个可以模拟任意多个集群的程序。

代码共有270行，非常适合用来模拟一组完整的集群程序，包括client、worker、代理、以及云端任务分发机制。

peering3: Full cluster simulation in C

```
//
//  同伴代理模拟（第三部分）
//  状态和任务消息流原型
//
//  示例程序使用了一个进程，这样可以让程序变得简单，
//  每个线程都有自己的上下文对象，所以可以认为他们是多个进程。
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define LRU_READY   "\001"      //  消息：worker已就绪

//  代理名称；现实中，这个名称应该由某种配置完成
static char *self;

//  请求-应答客户端使用REQ套接字
//  为模拟压力测试，客户端会一次性发送大量请求
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "ipc://%s-monitor.ipc", self);

    while (1) {
        sleep (randof (5));
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  使用随机的十六进制ID来标注任务
            zstr_send (client, task_id);

            //  最多等待10秒
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN,
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSE
            if (rc == -1)
                break;          //  中断

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
```

```
                    break;              //  中断
                //  worker的应答中应包含任务ID
                puts (reply);
                assert (streq (reply, task_id));
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: 客户端退出，丢失的任务为： %s", task_id);
                return NULL;
            }
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，并进行LRU路由
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告知代理worker已就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    while (1) {
        //  worker会随机延迟几秒
        zmsg_t *msg = zmsg_recv (worker);
        sleep (randof (2));
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是同伴代理的名称
    //
    if (argc < 2) {
        printf ("syntax: peering3 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));
```

```
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  将cloudfe绑定至端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  将statebe绑定至端点
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  将cloudbe连接至同伴代理的端点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的cloudfe端点\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }

    //  将statefe连接至同伴代理的端点
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的statebe端点\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  准备监控套接字
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);

    //  启动本地worker
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  启动本地client
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);

    //  有趣的部分
    //  -------------------------------------------------------------
```

```
    //  发布-订阅消息流
    //  - 轮询同伴代理的状态信息；
    //  - 当自身状态改变时，对外广播消息。
    //  请求-应答消息流
    //  - 若本地有可用worker，则轮询获取本地或云端的请求；
    //  - 将请求路由给本地worker或其他集群。

    //  可用worker队列
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t primary [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 },
            { statefe, 0, ZMQ_POLLIN, 0 },
            { monitor, 0, ZMQ_POLLIN, 0 }
        };
        //  如果没有可用的worker，则一直等待
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  中断

        //  跟踪自身状态信息是否改变
        int previous = local_capacity;

        //  处理本地worker的应答
        zmsg_t *msg = NULL;

        if (primary [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);
            local_capacity++;

            //  如果是“已就绪”的信号，则不再进行路由
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  处理来自同伴代理的应答
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  我们不需要使用同伴代理的地址
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
```

```
        }
        //  如果应答消息中的地址是同伴代理的，则发送给它
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  将应答路由给本地client
        if (msg)
            zmsg_send (&msg, localfe);

        //  处理同伴代理的状态更新
        if (primary [2].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (status);
        }
        //  处理监控消息
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }

        //  开始处理客户端请求
        //  - 如果本地有空闲worker，则总本地client和云端接收请求；
        //  - 如果我们只有空闲的同伴代理，则只轮询本地client的请求；
        //  - 将请求路由给本地worker，或者同伴代理。
        //
        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            if (local_capacity)
                rc = zmq_poll (secondary, 2, 0);
            else
                rc = zmq_poll (secondary, 1, 0);
            assert (rc >= 0);

            if (secondary [0].revents & ZMQ_POLLIN)
                msg = zmsg_recv (localfe);
            else
            if (secondary [1].revents & ZMQ_POLLIN)
                msg = zmsg_recv (cloudfe);
            else
                break;      //  没有任务

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
```

```
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  随机路由给同伴代理
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv
                zmsg_send (&msg, cloudbe);
            }
        }
        if (local_capacity != previous) {
            //  将自身代理的地址附加到消息中
            zstr_sendm (statebe, self);
            //  广播新的状态信息
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  程序结束后的清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

这段代码并不长，但花费了大约一天的时间去调通。以下是一些说明：

- client线程会检测并报告失败的请求，它们会轮询代理套接字，查看是否有应答，超时时间为10秒。

- client线程不会自己打印信息，而是将消息PUSH给一个监控线程，由它打印消息。这是我们第一次使用ZMQ进行监控和记录日志，我们以后会见得更多。

- clinet会模拟多种负载情况，让集群在不同的压力下工作，因此请求可能会在本地处理，也有可能会发送至云端。集群中的client和worker数量、其他集群的数量，以及延迟时间，会左右这个结果。你可以设置不同的参数来测试它们。

- 主循环中有两组轮询集合，事实上我们可以使用三个：信息流、后端、前端。因为在前面的例子中，如果后端没有空闲的worker，就没有必要轮询前端请求了。

以下是几个在编写过程中遇到的问题：

- 如果请求或应答在某处丢失，client会因此阻塞。回忆以下，ROUTER-ROUTER套接字会在消息如法路由的情况下直接丢弃。这里的一个策略就是改变client线程，检测并报告这种错误。此外，我还在每次recv()之后以及send()之前使用zmsg_dump()来打印套接字内容，用来更快地定位消息。

- 主循环会错误地从多个已就绪的套接字中获取消息，造成第一条消息的丢失。解决方法是只从第一个已就绪的套接字中获取消息。

- zmsg类库没有很好地将UUID编码为C语言字符串，导致包含字节0的UUID会崩溃。解决方法是将UUID转换成可打印的十六进制字符串。

这段模拟程序没有检测同伴代理是否存在。如果你开启了某个代理，它已向其他代理发送过状态信息，然后关闭了，那其他代理仍会向它发送请求。这样一来，其他代理的client就会报告很多错误。解决时有两点：一、为状态信息设置有效期，当同伴代理消失一段时间后就不再发送请求；二、提高请求-应答的可靠性，这在下一章中会讲到。

可靠的请求-应答模式

第三章中我们使用实例介绍了高级请求-应答模式，本章我们会讲述请求-应答模式的可靠性问题，并使用ZMQ提供的套接字类型组建起可靠的请求-应答消息系统。

本章将介绍的内容有：

- 客户端请求-应答
- 最近最少使用队列
- 心跳机制
- 面向服务的队列
- 基于磁盘（脱机）队列
- 主从备份服务
- 无中间件的请求-应答

什么是可靠性？

要给可靠性下定义，我们可以先界定它的相反面——故障。如果我们可以处理某些类型的故障，那么我们的模型对于这些故障就是可靠的。下面我们就来列举分布式ZMQ应用程序中可能发生的问题，从可能性高的故障开始：

- 应用程序代码是最大的故障来源。程序会崩溃或中止，停止对数据来源的响应，或是响应得太慢，耗尽内存等。
- 系统代码，如使用ZMQ编写的中间件，也会意外中止。系统代码应该要比应用程序代码更为可靠，但毕竟也有可能崩溃。特别是当系统代码与速度过慢的客户端交互时，很容易耗尽内存。
- 消息队列溢出，典型的情况是系统代码中没有对蛮客户端做积极的处理，任由消息队列溢出。
- 网络临时中断，造成消息丢失。这类错误ZMQ应用程序是无法及时发现的，因为ZMQ会自动进行重连。
- 硬件系统崩溃，导致所有进程中止。
- 网络会出现特殊情形的中断，如交换机的某个端口发生故障，导致部分网络无法访问。
- 数据中心可能遭受雷击、地震、火灾、电压过载、冷却系统失效等。

想要让软件系统规避上述所有的风险，需要大量的人力物力，故不在本指南的讨论范围之内。

由于前五个故障类型涵盖了99.9%的情形（这一数据源自我近期进行的一项研究），所以我们会深入探讨。如果你的公司大到足以考虑最后两种情形，那请及时联系我，因为我正愁没钱将我家后院的大坑建成游泳池。

可靠性设计

简单地来说，可靠性就是当程序发生故障时也能顺利地运行下去，这要比搭建一个消息系统来得困难得多。我们会根据ZMQ提供的每一种核心消息模式，来看看如何保障代码的持续运行。

- 请求-应答模式：当服务端在处理请求是中断，客户端能够得知这一信息，并停止接收消息，转而选择等待重试、请求另一服务端等操作。这里我们暂不讨论客户端发生问题的情形。

- 发布-订阅模式：如果客户端收到一些消息后意外中止，服务端是不知道这一情况的。发布-订阅模式中的订阅者不会返回任何消息给发布者。但是，订阅者可以通过其他方式联系服务端，如请求-应答模式，要求服务端重发消息。这里我们暂不讨论服务端发生问题的情形。此外，订阅者可以通过某些方式检查自身是否运行得过慢，并采取相应措施（向操作者发出警告、中止等）。

- 管道模式：如果worker意外终止，任务分发器将无从得知。管道模式和发布-订阅模式类似，只朝一个方向发送消息。但是，下游的结果收集器可以检测哪项任务没有完成，并告诉任务分发器重新分配该任务。如果任务分发器或结果收集器意外中止了，那客户端发出的请求只能另作处理。所以说，系统代码真的要减少出错的几率，因为这很难处理。

本章主要讲解请求-应答模式中的可靠性设计，其他模式将在后续章节中讲解。

最基本的请求应答模式是REQ客户端发送一个同步的请求至REP服务端，这种模式的可靠性很低。如果服务端在处理请求时中止，那客户端会永远处于等待状态。

相比TCP协议，ZMQ提供了自动重连机制、消息分发的负载均衡等。但是，在真实环境中这也是不够的。唯一可以完全信任基本请求-应答模式的应用场景是同一进程的两个线程之间进行通信，没有网络问题或服务器失效的情况。

但是，只要稍加修饰，这种基本的请求-应答模式就能很好地在现实环境中工作了。我喜欢将其称为“海盗”模式。

粗略地讲，客户端连接服务端有三种方式，每种方式都需要不同的可靠性设计：

- 多个客户端直接和单个服务端进行通信。使用场景：只有一个单点服务器，所有客户端都需要和它通信。需处理的故障：服务器崩溃和重启；网络连接中断。

- 多个客户端和单个队列装置通信，该装置将请求分发给多个服务端。使用场景：任务分发。需处理的故障：worker崩溃和重启，死循环，过载；队列装置崩溃和重启；网络中断。

- 多个客户端直接和多个服务端通信，无中间件。使用场景：类似域名解析的分布式服务。需处理的故障：服务端崩溃和重启，死循环，过载；网络连接中断。

以上每种设计都必须有所取舍，很多时候会混合使用。下面我们详细说明。

客户端的可靠性设计（懒惰海盗模式）

我们可以通过在客户端进行简单的设置，来实现可靠的请求-应答模式。我暂且称之为“懒惰的海盗”（Lazy Pirate）模式。

在接收应答时，我们不进行同步等待，而是做以下操作：

- 对REQ套接字进行轮询，当消息抵达时才进行接收；
- 请求超时后重发消息，循环多次；
- 若仍无消息，则结束当前事务。

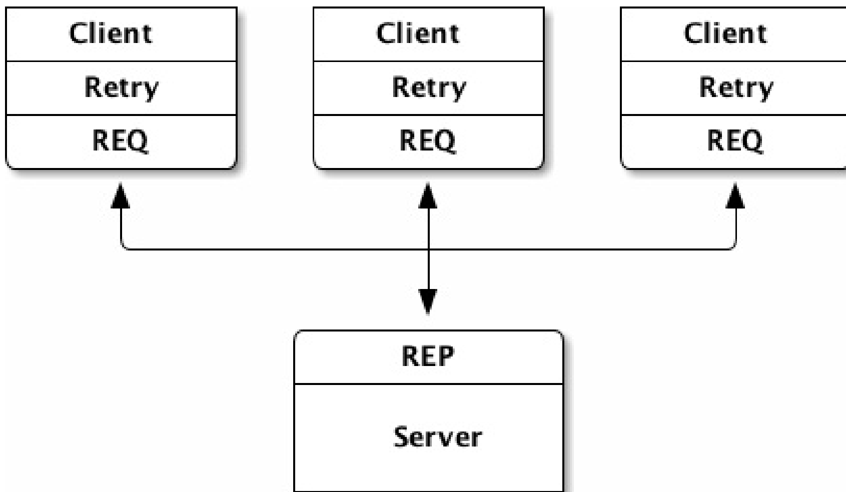


Figure 1 — Lazy Pirate pattern

使用REQ套接字时必须严格遵守发送-接收过程，因为它内部采用了一个有限状态机来限定状态，这一特性会让我们应用“海盗”模式时遇上一些麻烦。最简单的做法是将REQ套接字关闭重启，从而打破这一限定。

lpclient: Lazy Pirate client in C

```
//
//  Lazy Pirate client
//  使用zmq_poll轮询来实现安全的请求-应答
//  运行时可随机关闭或重启lpserver程序
//
#include "czmq.h"

#define REQUEST_TIMEOUT     2500    //  毫秒, (> 1000!)
#define REQUEST_RETRIES     3       //  尝试次数
#define SERVER_ENDPOINT     "tcp://localhost:5555"

int main (void)
{
    zctx_t *ctx = zctx_new ();
    printf ("I: 正在连接服务器...\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
```

```
    while (retries_left && !zctx_interrupted) {
        //  发送一个请求，并开始接收消息
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  对套接字进行轮询，并设置超时时间
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL
            if (rc == -1)
                break;          //  中断

            //  如果接收到回复则进行处理
            if (items [0].revents & ZMQ_POLLIN) {
                //  收到服务器应答，必须和请求时的序号一致
                char *reply = zstr_recv (client);
                if (!reply)
                    break;      //  Interrupted
                if (atoi (reply) == sequence) {
                    printf ("I: 服务器返回正常 (%s)\n", reply);
                    retries_left = REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                    printf ("E: 服务器返回异常: %s\n",
                        reply);

                free (reply);
            }
            else
            if (--retries_left == 0) {
                printf ("E: 服务器不可用，取消操作\n");
                break;
            }
            else {
                printf ("W: 服务器没有响应，正在重试...\n");
                //  关闭旧套接字，并建立新套接字
                zsocket_destroy (ctx, client);
                printf ("I: 服务器重连中...\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
                //  使用新套接字再次发送请求
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

lpserver: Lazy Pirate server in C

```
//
//  Lazy Pirate server
//  将REQ套接字连接至 tcp://*:5555
//  和hwserver程序类似，除了以下两点：
//   - 直接输出请求内容
//   - 随机地降慢运行速度，或中止程序，模拟崩溃
//
#include "zhelpers.h"

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_init (1);
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
        cycles++;

        //  循环几次后开始模拟各种故障
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: 模拟程序崩溃\n");
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: 模拟CPU过载\n");
            sleep (2);
        }
        printf ("I: 正常请求 (%s)\n", request);
        sleep (1);              //  耗时的处理过程
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_term (context);
    return 0;
}
```

运行这个测试用例时，可以打开两个控制台，服务端会随机发生故障，你可以看看客户端的反应。服务端的典型输出如下：

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

客户端的输出是：

```
I: connecting to server...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
W: no response from server, retrying...
I: connecting to server...
W: no response from server, retrying...
I: connecting to server...
E: server seems to be offline, abandoning
```

客户端为每次请求都加上了序列号，并检查收到的应答是否和序列号一致，以保证没有请求或应答丢失，同一个应答收到多次或乱序。多运行几次实例，看看是否真的能够解决问题。现实环境中你不需要使用到序列号，那只是为了证明这一方式是可行的。

客户端使用REQ套接字进行请求，并在发生问题时打开一个新的套接字来，绕过REQ强制的发送/接收过程。可能你会想用DEALER套接字，但这并不是一个好主意。首先，DEALER并不会像REQ那样处理信封（如果你不知道信封是什么，那更不能用DEALER了）。其次，你可能会获得你并不想得到的结果。

这一方案的优劣是：

- 优点：简单明了，容易实施；
- 优点：可以方便地应用到现有的客户端和服务端程序中；
- 优点：ZMQ有自动重连机制；
- 缺点：单点服务发生故障时不能定位到新的可用服务。

基本的可靠队列（简单海盗模式）

在第二种模式中，我们使用一个队列装置来扩展上述的“懒惰的海盗”模式，使客户端能够透明地和多个服务端通信。这里的服务端可以定义为worker。我们可以从最基础的模型开始，分阶段实施这个方案。

在所有的海盗模式中，worker是无状态的，或者说存在着一个我们所不知道的公共状态，如共享数据库。队列装置的存在意味着worker可以在client毫不知情的情况下随意进出。一个worker死亡后，会有另一个worker接替它的工作。这种拓扑结果非常简洁，但唯一的缺点是队列装置本身会难以维护，可能造成单点故障。

在第三章中，队列装置的基本算法是最近最少使用算法。那么，如果worker死亡或阻塞，我们需要做些什么？答案是很少很少。我们已经在client中加入了重试的机制，所以，使用基本的LRU队列就可以运作得很好了。这种做法也符合ZMQ的逻辑，所以我们可以通过在点对点交互中插入一个简单的队列装置来扩展它：

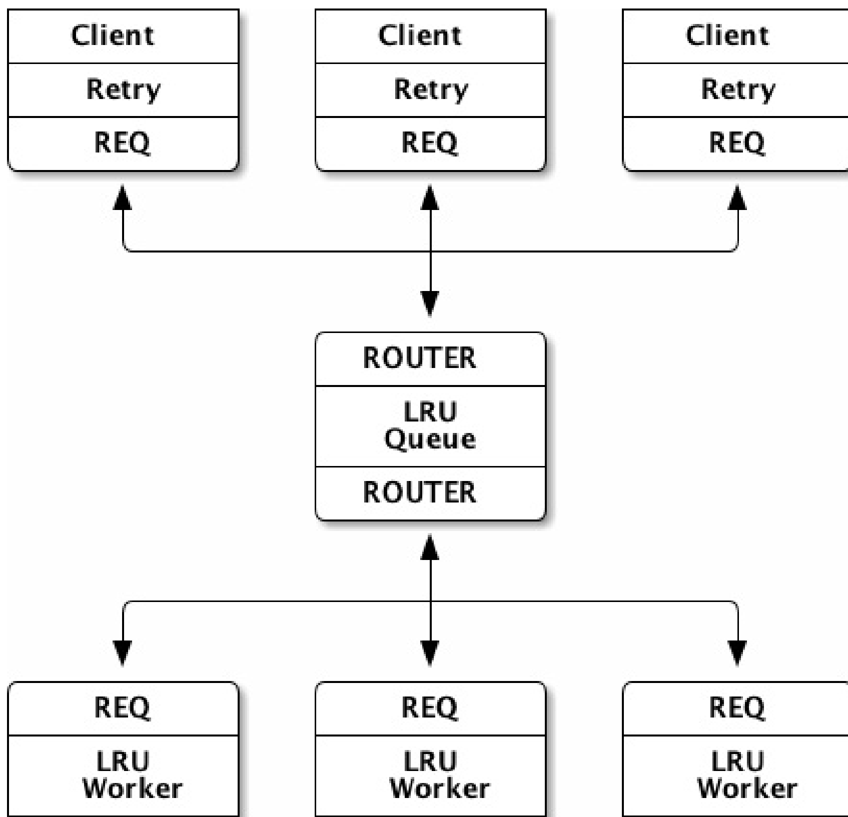


Figure 2 — Simple Pirate Pattern

我们可以直接使用“懒惰的海盗”模式中的client，以下是队列装置的代码：

spqueue: Simple Pirate queue in C

```
//
//  简单海盗队列
//
//  这个装置和LRU队列完全一致，不存在任何可靠性机制，依靠client的重试来保证装
//
#include "czmq.h"

#define LRU_READY   "\001"      //  消息：worker准备就绪

int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  client端点
    zsocket_bind (backend,  "tcp://*:5556");    //  worker端点
```

```
    //  存放可用worker的队列
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当有可用的woker时，轮询前端端点
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  中断

        //  处理后端端点的worker消息
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker的地址进行LRU排队
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  如果消息不是READY，则转发给client
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取client请求，转发给第一个可用的worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  程序运行结束，进行清理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

以下是worker的代码，用到了“懒惰的海盗”服务，并将其调整为LRU模式（使用REQ套接字传递“已就绪”信号）：

spworker: Simple Pirate worker in C

```
//
//  简单海盗模式worker
//
//  使用REQ套接字连接tcp://*:5556，使用LRU算法实现worker
//
#include "czmq.h"
#define LRU_READY   "\001"      //  消息：worker已就绪

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    //  使用随机符号来指定套接字标识，方便追踪
    srandom ((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10(
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen (identit
    zsocket_connect (worker, "tcp://localhost:5556");

    //  告诉代理worker已就绪
    printf ("I: (%s) worker准备就绪\n", identity);
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断

        //  经过几轮循环后，模拟各种问题
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) 模拟崩溃\n", identity);
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) 模拟CPU过载\n", identity);
            sleep (3);
            if (zctx_interrupted)
                break;
        }
        printf ("I: (%s) 正常应答\n", identity);
        sleep (1);              //  进行某些处理
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
```

```
    return 0;
}
```

运行上述事例，启动多个worker，一个client，以及一个队列装置，顺序随意。你可以看到worker最终都会崩溃或死亡，client则多次重试并最终放弃。装置从来不会停止，你可以任意重启worker和client，这个模型可以和任意个worker、client交互。

健壮的可靠队列（偏执海盗模式）

“简单海盗队列”模式工作得非常好，主要是因为它只是两个现有模式的结合体。不过，它也有一些缺点：

- 该模式无法处理队列的崩溃或重启。client会进行重试，但worker不会重启。虽然ZMQ会自动重连worker的套接字，但对于新启动的队列装置来说，由于worker并没有发送“已就绪”的消息，所以它相当于是不存在的。为了解决这一问题，我们需要从队列发送心跳给worker，这样worker就能知道队列是否已经死亡。

- 队列没有检测worker是否已经死亡，所以当worker在处于空闲状态时死亡，队列装置只有在发送了某个请求之后才会将该worker从队列中移除。这时，client什么都不能做，只能等待。这不是一个致命的问题，但是依然是不够好的。所以，我们需要从worker发送心跳给队列装置，从而让队列得知worker什么时候消亡。

我们使用一个名为“偏执的海盗模式”来解决上述两个问题。

之前我们使用REQ套接字作为worker的套接字类型，但在偏执海盗模式中我们会改用DEALER套接字，从而使我们能够任意地发送和接受消息，而不是像REQ套接字那样必须完成发送-接受循环。而DEALER的缺点是我们必须自己管理消息信封。如果你不知道信封是什么，那请阅读第三章。

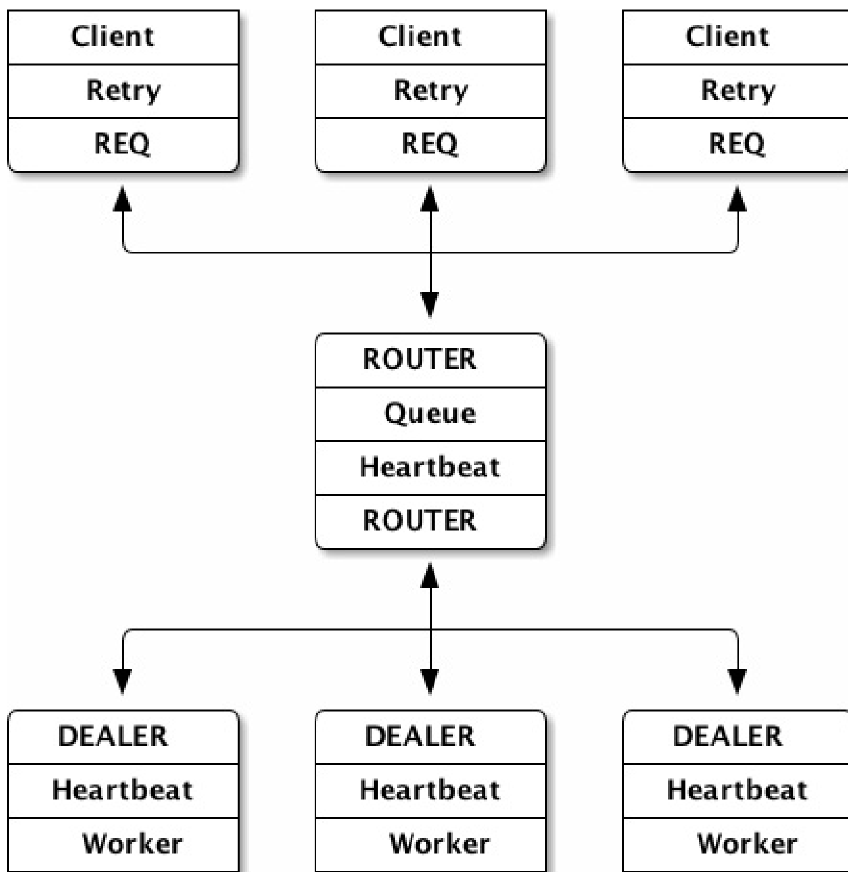


Figure 3 — Paranoid Pirate Pattern

我们仍会使用懒惰海盗模式的client，以下是偏执海盗的队列装置代码：

ppqueue: Paranoid Pirate queue in C

```
//
//  偏执海盗队列
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  心跳健康度，3-5是合理的
#define HEARTBEAT_INTERVAL  1000    //  单位：毫秒

//  偏执海盗协议的消息代码
#define PPP_READY       "\001"      //  worker已就绪
#define PPP_HEARTBEAT   "\002"      //  worker心跳


//  使用以下结构表示worker队列中的一个有效的worker

typedef struct {
    zframe_t *address;          //  worker的地址
    char *identity;             //  可打印的套接字标识
    int64_t expiry;             //  过期时间
} worker_t;
```

```
//  创建新的worker
static worker_t *
s_worker_new (zframe_t *address)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->address = address;
    self->identity = zframe_strdup (address);
    self->expiry = zclock_time () + HEARTBEAT_INTERVAL * HEARTBEAT_
    return self;
}

//  销毁worker结构，包括标识
static void
s_worker_destroy (worker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->address);
        free (self->identity);
        free (self);
        *self_p = NULL;
    }
}

//  worker已就绪，将其移至列表末尾
static void
s_worker_ready (worker_t *self, zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->identity, worker->identity)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
    zlist_append (workers, self);
}

//  返回下一个可用的worker地址
static zframe_t *
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->address;
    worker->address = NULL;
    s_worker_destroy (&worker);
    return frame;
}
```

```
//  寻找并销毁已过期的worker。
//  由于列表中最旧的worker排在最前，所以当找到第一个未过期的worker时就停止。
static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  worker未过期，停止扫描

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}


int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend  = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  client端点
    zsocket_bind (backend,  "tcp://*:5556");    //  worker端点
    //  List of available workers
    zlist_t *workers = zlist_new ();

    //  规律地发送心跳
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当存在可用worker时轮询前端端点
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
            HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  处理后端worker请求
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker地址进行LRU路由
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断

            //  worker的任何信号均表示其仍然存活
            zframe_t *address = zmsg_unwrap (msg);
            worker_t *worker = s_worker_new (address);
            s_worker_ready (worker, workers);
```

```
                //  处理控制消息，或者将应答转发给client
                if (zmsg_size (msg) == 1) {
                    zframe_t *frame = zmsg_first (msg);
                    if (memcmp (zframe_data (frame), PPP_READY, 1)
                    &&  memcmp (zframe_data (frame), PPP_HEARTBEAT, 1))
                        printf ("E: invalid message from worker");
                        zmsg_dump (msg);
                    }
                    zmsg_destroy (&msg);
                }
                else
                    zmsg_send (&msg, frontend);
            }
            if (items [1].revents & ZMQ_POLLIN) {
                //  获取下一个client请求，交给下一个可用的worker
                zmsg_t *msg = zmsg_recv (frontend);
                if (!msg)
                    break;          //  中断
                zmsg_push (msg, s_workers_next (workers));
                zmsg_send (&msg, backend);
            }

            //  发送心跳给空闲的worker
            if (zclock_time () >= heartbeat_at) {
                worker_t *worker = (worker_t *) zlist_first (workers);
                while (worker) {
                    zframe_send (&worker->address, backend,
                                 ZFRAME_REUSE + ZFRAME_MORE);
                    zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
                    zframe_send (&frame, backend, 0);
                    worker = (worker_t *) zlist_next (workers);
                }
                heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            }
            s_workers_purge (workers);
        }

        //  程序结束后进行清理
        while (zlist_size (workers)) {
            worker_t *worker = (worker_t *) zlist_pop (workers);
            s_worker_destroy (&worker);
        }
        zlist_destroy (&workers);
        zctx_destroy (&ctx);
        return 0;
    }
```

该队列装置使用心跳机制扩展了LRU模式，看起来很简单，但要想出这个主意还挺难的。下文会更多地介绍心跳机制。

以下是偏执海盗的worker代码：

ppworker: Paranoid Pirate worker in C

```
//
//  偏执海盗worker
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  合理值：3-5
#define HEARTBEAT_INTERVAL  1000    //  单位：毫秒
#define INTERVAL_INIT       1000    //  重试间隔
#define INTERVAL_MAX       32000    //  回退算法最大值

//  偏执海盗规范的常量定义
#define PPP_READY       "\001"      //  消息：worker已就绪
#define PPP_HEARTBEAT   "\002"      //  消息：worker心跳

//  返回一个连接至偏执海盗队列装置的套接字

static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    //  告知队列worker已准备就绪
    printf ("I: worker已就绪\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    //  如果心跳健康度为零，则表示队列装置已死亡
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    //  规律地发送心跳
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    srandom ((unsigned) time (NULL));
    int cycles = 0;
    while (1) {
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } 
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
```

```
            //  获取消息
            //  - 3段消息，信封+内容，表示一个请求
            //  - 1段消息，表示心跳
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          //  中断

            if (zmsg_size (msg) == 3) {
                //  若干词循环后模拟各种问题
                cycles++;
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: 模拟崩溃\n");
                    zmsg_destroy (&msg);
                    break;
                }
                else
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: 模拟CPU过载\n");
                    sleep (3);
                    if (zctx_interrupted)
                        break;
                }
                printf ("I: 正常应答\n");
                zmsg_send (&msg, worker);
                liveness = HEARTBEAT_LIVENESS;
                sleep (1);              //  做一些处理工作
                if (zctx_interrupted)
                    break;
            }
            else
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)
                    liveness = HEARTBEAT_LIVENESS;
                else {
                    printf ("E: 非法消息\n");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else {
                printf ("E: 非法消息\n");
                zmsg_dump (msg);
            }
            interval = INTERVAL_INIT;
        }
        else
        if (--liveness == 0) {
            printf ("W: 心跳失败，无法连接队列装置\n");
            printf ("W: %zd 毫秒后进行重连...\n", interval);
            zclock_sleep (interval);

            if (interval < INTERVAL_MAX)
```

```
                interval *= 2;
            zsocket_destroy (ctx, worker);
            worker = s_worker_socket (ctx);
            liveness = HEARTBEAT_LIVENESS;
        }

        //  适时发送心跳给队列
        if (zclock_time () > heartbeat_at) {
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            printf ("I: worker心跳\n");
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 代码中包含了几处失败模拟，和先前一样。这会让代码极难维护，所以当投入使用时，应当移除这些模拟代码。
- 偏执海盗模式中队列的心跳有时会不正常，下文会讲述这一点。
- worker使用了一种类似于懒惰海盗client的重试机制，但有两点不同：1、回退算法设置；2、永不言弃。

尝试运行以下代码，跑通流程：

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

你会看到worker逐个崩溃，client在多次尝试后放弃。你可以停止并重启队列装置，client和worker会相继重连，并正确地发送、处理和接收请求，顺序不会混乱。所以说，整个通信过程只有两种情形：交互成功，或client最终放弃。

心跳

当我在写偏执海盗模式的示例时，大约花了五个小时的时间来协调队列至worker的心跳，剩下的请求-应答链路只花了约10分钟的时间。心跳机制在可靠性上带来的益处有时还不及它所引发的问题。使用过程中很有可能会产生“虚假故障”的情况，即节点误认为他们已失去连接，因为心跳没有正确地发送。

在理解和实施心跳时，需要考虑以下几点：

- 心跳不是一种请求-应答，它们异步地在节点之间传递，任一节点都可以通过它来判断对方已经死亡，并中止通信。

- 如果某个节点使用持久套接字（即设定了套接字标识），意味着发送给它的心跳可能会堆砌，并在重连后一起收到。所以说，worker不应该使用持久套接字。示例代码使用持久套接字是为了便于调试，而且代码中使用了随机的套接字标识，避免重用之前的标识。

- 使用过程中，应先让心跳工作起来，再进行后面的消息处理。你需要保证启动任一节点后，心跳都能正确地执行。停止并重启他们，模拟冻结、崩溃等情况来进行测试。

- 当你的主循环使用了zmq_poll()，则应该使用另一个计时器来触发心跳。不要使用主循环来控制心跳的发送，这回导致过量地发送心跳（阻塞网络），或是发送得太少（导致节点断开）。zhelpers包提供了s_clock()函数返回当前系统时间戳，单位是毫秒，可以用它来控制心跳的发送间隔。C代码如下：

```
// 规律地发送心跳
uint64_t heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
while (1) {
    …
    zmq_poll (items, 1, HEARTBEAT_INTERVAL * 1000);
    …
    // 无论zmq_poll的行为是什么，都使用以下逻辑判断是否发送心跳
    if (s_clock () > heartbeat_at) {
        … 发送心跳给所有节点
        // 设置下一次心跳的时间
        heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
    }
}
```

- 主循环应该使用心跳间隔作为超时时间。显然不能使用无超时时间的设置，而短于心跳间隔也只是浪费循环次数而已。

- 使用简单的追踪方式来进行追踪，如直接输出至控制台。这里有一些追踪的窍门：使用zmsg()函数打印套接字内容；对消息进行编号，判断是否会有间隔。

- 在真实的应用程序中，心跳必须是可以配置的，并能和节点共同商定。有些节点需要高频心跳，如10毫秒，另一些节点则可能只需要30秒发送一次心跳即可。

- 如果你要对不同的节点发送不同频率的心跳，那么poll的超时时间应设置为最短的心跳间隔。

- 也许你会想要用一个单独的套接字来处理心跳，这看起来很棒，可以将同步的请求-应答和异步的心跳隔离开来。但是，这个主意并不好，原因有几点：首先、发送数据时其实是不需要发送心跳的；其次、套接字可能会因为网络问题而阻塞，你需要设法知道用于发送数据的套接字停止响应的原因是死亡了还是过于繁忙而已，这样你就需要对这个套接字进行心跳。最后，处理两个套接字要比处理一个复杂得多。

- 我们没有设置client至队列的心跳，因为这太过复杂了，而且没有太大价值。

约定和协议

也许你已经注意到，由于心跳机制，偏执海盗模式和简单海盗模式是不兼容的。

其实，这里我们需要写一个协议。也许在试验阶段是不需要协议的，但这在真实的应用程序中是有必要。如果我们想用其他语言来写worker怎么办？我们是否需要通过源代码来查看通信过程？如果我们想改变协议怎么办？规范可能很简单，但并不显然。越是成功的协议，就会越为复杂。

一个缺乏约定的应用程序一定是不可复用的，所以让我们来为这个协议写一个规范，怎么做呢？

- 位于rfc.zeromq.org的wiki页上，我们特地设置了一个用于存放ZMQ协议的页面。
    - 要创建一个新的协议，你需要注册并按照指导进行。过程很直接，但并不一定所有人都能撰写技术性文档。

我大约花了15分钟的时间草拟海盗模式规范（PPP），麻雀虽小，但五脏俱全。

要用PPP协议进行真实环境下的编程，你还需要：

- 在READY命令中加入版本号，这样就能再日后安全地新增PPP版本号。
- 目前，READY和HEARTBEAT信号并没有指定其来源于请求还是应答。要区分他们，需要新建一个消息结构，其中包含“消息类型”这一信息。

面向服务的可靠队列（管家模式）

世上的事物往往瞬息万变，正当我们期待有更好的协议来解决上一节的问题时，已经有人制定好了：

- http://rfc.zeromq.org/spec:7

这份协议只有一页，它将PPP协议变得更为坚固。我们在设计复杂架构时应该这样做：首先写下约定，再用软件去实现它。

管家模式协议（MDP）在扩展PPP协议时引入了一个有趣的特性：client发送的每一个请求都有一个“服务名称”，而worker在像队列装置注册时需要告知自己的服务类型。MDP的优势在于它来源于现实编程，协议简单，且容易提升。

引入“服务名称”的机制，是对偏执海盗队列的一个简单补充，而结果是让其成为一个面向服务的代理。

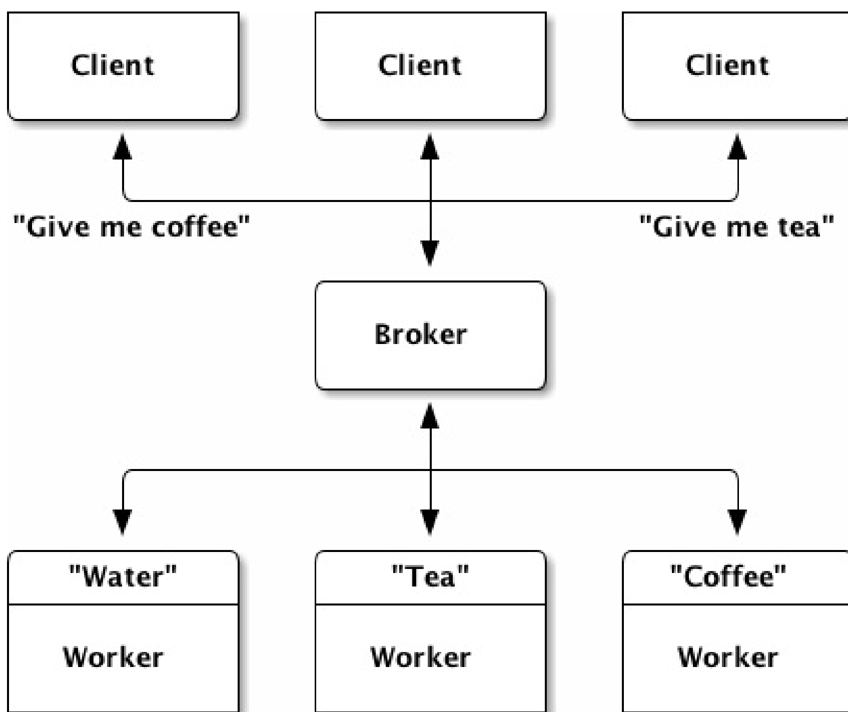


Figure 4 — Majordomo Pattern

在实施管家模式之前，我们需要为client和worker编写一个框架。如果程序员可以通过简单的API来实现这种模式，那就没有必要让他们去了解管家模式的协议内容和实现方法了。 所以，我们第一个协议（即管家模式协议）定义了分布式架构中节点是如何互相交互的，第二个协议则要定义应用程序应该如何通过框架来使用这一协议。 管家模式有两个端点，客户端和服务端。因为我们要为client和worker都撰写框架，所以就需要提供两套API。以下是用简单的面向对象方法设计的client端API雏形，使用的是C语言的ZFL library。

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
zmsg_t  *mdcli_send    (mdcli_t *self, char *service, zmsg_t **requ
```

就这么简单。我们创建了一个会话来和代理通信，发送并接收一个请求，最后关闭连接。以下是worker端API的雏形。

```
mdwrk_t *mdwrk_new     (char *broker,char *service);
void     mdwrk_destroy (mdwrk_t **self_p);
zmsg_t  *mdwrk_recv    (mdwrk_t *self, zmsg_t *reply);
```

上面两段代码看起来差不多，但是worker端API略有不同。worker第一次执行recv()后会传递一个空的应答，之后才传递当前的应答，并获得新的请求。

两段的API都很容易开发，只需在偏执海盗模式代码的基础上修改即可。以下是client API：

mdcliapi: Majordomo client API in C

```
/*  =====================================================================
    mdcliapi.c

    Majordomo Protocol Client API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "mdcliapi.h"

//  类结构
//  我们会通过成员方法来访问这些属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  连接至代理的套接字
    int verbose;                //  使用标准输出打印当前活动
    int timeout;                //  请求超时时间
    int retries;                //  请求重试次数
};


//  ---------------------------------------------------------------------
//  连接或重连代理

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_REQ);
    zmq_connect (self->client, self->broker);
```

```
    if (self->verbose)
        zclock_log ("I: 正在连接至代理 %s...", self->broker);
}


//  ---------------------------------------------------------------
//  构造函数

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  毫秒
    self->retries = 3;              //  尝试次数

    s_mdcli_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------
//  析构函数

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------
//  设定请求超时时间

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}
```

```
//  ---------------------------------------------------------------
//  设定请求重试次数

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}


//  ---------------------------------------------------------------
//  向代理发送请求，并尝试获取应答；
//  对消息保持所有权，发送后销毁；
//  返回应答消息，或NULL。

zmsg_t *
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  用协议前缀包装消息
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: 服务名称 (可打印字符串)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    if (self->verbose) {
        zclock_log ("I: 发送请求给 '%s' 服务:", service);
        zmsg_dump (request);
    }

    int retries_left = self->retries;
    while (retries_left && !zctx_interrupted) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->client);

        while (TRUE) {
            //  轮询套接字以接收应答，有超时时间
            zmq_pollitem_t items [] = {
                { self->client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_M
            if (rc == -1)
                break;          //  中断

            //  收到应答后进行处理
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (self->client);
                if (self->verbose) {
                    zclock_log ("I: received reply:");
                    zmsg_dump (msg);
                }
```

```
                //  不要尝试处理错误，直接报错即可
                assert (zmsg_size (msg) >= 3);

                zframe_t *header = zmsg_pop (msg);
                assert (zframe_streq (header, MDPC_CLIENT));
                zframe_destroy (&header);

                zframe_t *reply_service = zmsg_pop (msg);
                assert (zframe_streq (reply_service, service));
                zframe_destroy (&reply_service);

                zmsg_destroy (&request);
                return msg;     //  成功
            }
            else
            if (--retries_left) {
                if (self->verbose)
                    zclock_log ("W: no reply, reconnecting...");
                //  重连并重发消息
                s_mdcli_connect_to_broker (self);
                zmsg_t *msg = zmsg_dup (request);
                zmsg_send (&msg, self->client);
            }
            else {
                if (self->verbose)
                    zclock_log ("W: 发生严重错误，放弃重试。");
                break;          //  放弃
            }
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，结束client进程...\n");
    zmsg_destroy (&request);
    return NULL;
}
```

以下测试程序会执行10万次请求应答：

mdclient: Majordomo client application in C

```
//
//  管家模式协议 - 客户端示例
//  使用mdcli API隐藏管家模式协议的内部实现
//

//  让我们直接编译这段代码，不生成类库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        zmsg_t *reply = mdcli_send (session, "echo", &request);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  中断或停止
    }
    printf ("已处理 %d 次请求-应答\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

下面是worker的API：

mdwrkapi: Majordomo worker API in C

```
/*  ===================================================================
    mdwrkapi.c

    Majordomo Protocol Worker API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7

    -------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
```

```
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    ===============================================================
*/

#include "mdwrkapi.h"

//  可靠性参数
#define HEARTBEAT_LIVENESS  3       //  合理值：3-5

//  类结构
//  使用成员函数访问属性

struct _mdwrk_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    char *service;
    void *worker;               //  连接至代理的套接字
    int verbose;                //  使用标准输出打印活动

    //  心跳设置
    uint64_t heartbeat_at;      //  发送心跳的时间
    size_t liveness;            //  尝试次数
    int heartbeat;              //  心跳延时，单位：毫秒
    int reconnect;              //  重连延时，单位：毫秒

    //  内部状态
    int expect_reply;           //  初始值为0

    //  应答地址，如果存在的话
    zframe_t *reply_to;
};


//  ---------------------------------------------------------------
//  发送消息给代理
//  如果没有提供消息，则内部创建一个

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  将协议信封压入消息顶部
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
```

```
    zmsg_pushstr (msg, MDPW_WORKER);
    zmsg_pushstr (msg, "");

    if (self->verbose) {
        zclock_log ("I: sending %s to broker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->worker);
}


//  ---------------------------------------------------------------------
//  连接或重连代理

void s_mdwrk_connect_to_broker (mdwrk_t *self)
{
    if (self->worker)
        zsocket_destroy (self->ctx, self->worker);
    self->worker = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->worker, self->broker);
    if (self->verbose)
        zclock_log ("I: 正在连接代理 %s...", self->broker);

    //  向代理注册服务类型
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    //  当心跳健康度为零，表示代理已断开连接
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
}


//  ---------------------------------------------------------------------
//  构造函数

mdwrk_t *
mdwrk_new (char *broker,char *service, int verbose)
{
    assert (broker);
    assert (service);

    mdwrk_t *self = (mdwrk_t *) zmalloc (sizeof (mdwrk_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->service = strdup (service);
    self->verbose = verbose;
    self->heartbeat = 2500;     //  毫秒
    self->reconnect = 2500;     //  毫秒

    s_mdwrk_connect_to_broker (self);
    return self;
}
```

```
//  ---------------------------------------------------------------------
//  析构函数

void
mdwrk_destroy (mdwrk_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdwrk_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self->service);
        free (self);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  设置心跳延迟

void
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
    self->heartbeat = heartbeat;
}


//  ---------------------------------------------------------------------
//  设置重连延迟

void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}


//  ---------------------------------------------------------------------
//  若有应答则发送给代理，并等待新的请求

zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    //  格式化并发送请求传入的应答
    assert (reply_p);
    zmsg_t *reply = *reply_p;
    assert (reply || !self->expect_reply);
    if (reply) {
        assert (self->reply_to);
        zmsg_wrap (reply, self->reply_to);
```

```
        s_mdwrk_send_to_broker (self, MDPW_REPLY, NULL, reply);
        zmsg_destroy (reply_p);
    }
    self->expect_reply = 1;

    while (TRUE) {
        zmq_pollitem_t items [] = {
            { self->worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, self->heartbeat * ZMQ_POLL_MSE
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
                break;          //  中断
            if (self->verbose) {
                zclock_log ("I: 从代理处获得消息:");
                zmsg_dump (msg);
            }
            self->liveness = HEARTBEAT_LIVENESS;

            //  不要处理错误，直接报错即可
            assert (zmsg_size (msg) >= 3);

            zframe_t *empty = zmsg_pop (msg);
            assert (zframe_streq (empty, ""));
            zframe_destroy (&empty);

            zframe_t *header = zmsg_pop (msg);
            assert (zframe_streq (header, MDPW_WORKER));
            zframe_destroy (&header);

            zframe_t *command = zmsg_pop (msg);
            if (zframe_streq (command, MDPW_REQUEST)) {
                //  这里需要将消息中空帧之前的所有地址都保存起来，
                //  但在这里我们暂时只保存一个
                self->reply_to = zmsg_unwrap (msg);
                zframe_destroy (&command);
                return msg;     //  处理请求
            }
            else
            if (zframe_streq (command, MDPW_HEARTBEAT))
                ;               //  不对心跳做任何处理
            else
            if (zframe_streq (command, MDPW_DISCONNECT))
                s_mdwrk_connect_to_broker (self);
            else {
                zclock_log ("E: 消息不合法");
                zmsg_dump (msg);
            }
            zframe_destroy (&command);
            zmsg_destroy (&msg);
```

```
        }
        else
        if (--self->liveness == 0) {
            if (self->verbose)
                zclock_log ("W: 失去与代理的连接 - 正在重试...");
            zclock_sleep (self->reconnect);
            s_mdwrk_connect_to_broker (self);
        }
        //  适时地发送消息
        if (zclock_time () > self->heartbeat_at) {
            s_mdwrk_send_to_broker (self, MDPW_HEARTBEAT, NULL, NUL
            self->heartbeat_at = zclock_time () + self->heartbeat;
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，中止worker...\n");
    return NULL;
}
```

以下测试程序实现了名为echo的服务：

mdworker: Majordomo worker application in C

```
//
//  管家模式协议 - worker示例
//  使用mdwrk API隐藏MDP协议的内部实现
//

//  让我们直接编译代码，而不创建类库
#include "mdwrkapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdwrk_t *session = mdwrk_new (
        "tcp://localhost:5555", "echo", verbose);

    zmsg_t *reply = NULL;
    while (1) {
        zmsg_t *request = mdwrk_recv (session, &reply);
        if (request == NULL)
            break;              //  worker被中止
        reply = request;        //  echo服务……其实很复杂:)
    }
    mdwrk_destroy (&session);
    return 0;
}
```

几点说明：

- API是单线程的，所以说worker不会再后台发送心跳，而这也是我们所期望的：如果worker应用程序停止了，心跳就会跟着中止，代理便会停止向该worker发送新的请求。

- wroker API没有做回退算法的设置，因为这里不值得使用这一复杂的机制。

- API没有提供任何报错机制，如果出现问题，它会直接报断言（或异常，依语言而定）。这一做法对实验性的编程是有用的，这样可以立刻看到执行结果。但在真实编程环境中，API应该足够健壮，合适地处理非法消息。

也许你会问，worker API为什么要关闭它的套接字并新开一个呢？特别是ZMQ是有重连机制的，能够在节点归来后进行重连。我们可以回顾一下简单海盗模式中的worker，以及偏执海盗模式中的worker来加以理解。ZMQ确实会进行自动重连，但如果代理死亡并重连，worker并不会重新进行注册。这个问题有两种解决方案：一是我们这里用到的较为简便的方案，即当worker判断代理已经死亡时，关闭它的套接字并重头来过；另一个方案是当代理收到未知worker的心跳时要求该worker对其提供的服务类型进行注册，这样一来就需要在协议中说明这一规则。

下面让我们设计管家模式的代理，它的核心代码是一组队列，每种服务对应一个队列。我们会在worker出现时创建相应的队列（worker消失时应该销毁对应的队列，不过我们这里暂时不考虑）。额外的，我们会为每种服务维护一个worker的队列。

为了让C语言代码更为易读易写，我使用了ZFL项目提供的哈希和链表容器，并命名为zhash和zlist。如果使用现代语言编写，那自然可以使用其内置的容器。

mdbroker: Majordomo broker in C

```
//
//  管家模式协议 - 代理
//  协议 http://rfc.zeromq.org/spec:7 和 spec:8 的最简实现
//
#include "czmq.h"
#include "mdp.h"

//  一般我们会从配置文件中获取以下值

#define HEARTBEAT_LIVENESS  3       //  合理值：3-5
#define HEARTBEAT_INTERVAL  2500    //  单位：毫秒
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

//  定义一个代理
typedef struct {
    zctx_t *ctx;                //  上下文
    void *socket;               //  用于连接client和worker的套接字
    int verbose;                //  使用标准输出打印活动信息
    char *endpoint;             //  代理绑定到的端点
    zhash_t *services;          //  已知服务的哈希表
    zhash_t *workers;           //  已知worker的哈希表
    zlist_t *waiting;           //  正在等待的worker队列
    uint64_t heartbeat_at;      //  发送心跳的时间
} broker_t;
```

```
//  定义一个服务
typedef struct {
    char *name;             //  服务名称
    zlist_t *requests;      //  客户端请求队列
    zlist_t *waiting;       //  正在等待的worker队列
    size_t workers;         //  可用worker数
} service_t;

//  定义一个worker，状态为空闲或占用
typedef struct {
    char *identity;         //  worker的标识
    zframe_t *address;      //  地址帧
    service_t *service;     //  所属服务
    int64_t expiry;         //  过期时间，从未收到心跳起计时
} worker_t;


//  ---------------------------------------------------------------
//  代理使用的函数
static broker_t *
    s_broker_new (int verbose);
static void
    s_broker_destroy (broker_t **self_p);
static void
    s_broker_bind (broker_t *self, char *endpoint);
static void
    s_broker_purge_workers (broker_t *self);

//  服务使用的函数
static service_t *
    s_service_require (broker_t *self, zframe_t *service_frame);
static void
    s_service_destroy (void *argument);
static void
    s_service_dispatch (broker_t *self, service_t *service, zmsg_t
static void
    s_service_internal (broker_t *self, zframe_t *service_frame, zr

//  worker使用的函数
static worker_t *
    s_worker_require (broker_t *self, zframe_t *address);
static void
    s_worker_delete (broker_t *self, worker_t *worker, int disconne
static void
    s_worker_destroy (void *argument);
static void
    s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg
static void
    s_worker_send (broker_t *self, worker_t *worker, char *command,
                   char *option, zmsg_t *msg);
static void
    s_worker_waiting (broker_t *self, worker_t *worker);
```

```
//  客户端使用的函数
static void
    s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg


//  ---------------------------------------------------------------------
//  主程序

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    broker_t *self = s_broker_new (verbose);
    s_broker_bind (self, "tcp://*:5555");

    //  接受并处理消息，直至程序被中止
    while (TRUE) {
        zmq_pollitem_t items [] = {
            { self->socket,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_
        if (rc == -1)
            break;              //  中断

        //  Process next input message, if any
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          //  中断
            if (self->verbose) {
                zclock_log ("I: 收到消息:");
                zmsg_dump (msg);
            }
            zframe_t *sender = zmsg_pop (msg);
            zframe_t *empty  = zmsg_pop (msg);
            zframe_t *header = zmsg_pop (msg);

            if (zframe_streq (header, MDPC_CLIENT))
                s_client_process (self, sender, msg);
            else
            if (zframe_streq (header, MDPW_WORKER))
                s_worker_process (self, sender, msg);
            else {
                zclock_log ("E: 非法消息:");
                zmsg_dump (msg);
                zmsg_destroy (&msg);
            }
            zframe_destroy (&sender);
            zframe_destroy (&empty);
            zframe_destroy (&header);
        }
        //  断开并删除过期的worker
        //  适时地发送心跳给worker
        if (zclock_time () > self->heartbeat_at) {
```

```
            s_broker_purge_workers (self);
            worker_t *worker = (worker_t *) zlist_first (self->wait
            while (worker) {
                s_worker_send (self, worker, MDPW_HEARTBEAT, NULL,
                worker = (worker_t *) zlist_next (self->waiting);
            }
            self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVA
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，关闭中...\n");

    s_broker_destroy (&self);
    return 0;
}


//  ---------------------------------------------------------------------
//  代理对象的构造函数

static broker_t *
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    //  初始化代理状态
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->verbose = verbose;
    self->services = zhash_new ();
    self->workers = zhash_new ();
    self->waiting = zlist_new ();
    self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    return self;
}

//  ---------------------------------------------------------------------
//  代理对象的析构函数

static void
s_broker_destroy (broker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        broker_t *self = *self_p;
        zctx_destroy (&self->ctx);
        zhash_destroy (&self->services);
        zhash_destroy (&self->workers);
        zlist_destroy (&self->waiting);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  ---------------------------------------------------------------------
//  将代理套接字绑定至端点，可以重复调用该函数
//  我们使用一个套接字来同时处理client和worker

void
s_broker_bind (broker_t *self, char *endpoint)
{
    zsocket_bind (self->socket, endpoint);
    zclock_log ("I: MDP broker/0.1.1 is active at %s", endpoint);
}

//  ---------------------------------------------------------------------
//  删除空闲状态中过期的worker

static void
s_broker_purge_workers (broker_t *self)
{
    worker_t *worker = (worker_t *) zlist_first (self->waiting);
    while (worker) {
        if (zclock_time () < worker->expiry)
            continue;               //  该worker未过期，停止搜索
        if (self->verbose)
            zclock_log ("I: 正在删除过期的worker: %s",
                worker->identity);

        s_worker_delete (self, worker, 0);
        worker = (worker_t *) zlist_first (self->waiting);
    }
}

//  ---------------------------------------------------------------------
//  定位或创建新的服务项

static service_t *
s_service_require (broker_t *self, zframe_t *service_frame)
{
    assert (service_frame);
    char *name = zframe_strdup (service_frame);

    service_t *service =
        (service_t *) zhash_lookup (self->services, name);
    if (service == NULL) {
        service = (service_t *) zmalloc (sizeof (service_t));
        service->name = name;
        service->requests = zlist_new ();
        service->waiting = zlist_new ();
        zhash_insert (self->services, name, service);
        zhash_freefn (self->services, name, s_service_destroy);
        if (self->verbose)
            zclock_log ("I: 收到消息:");
    }
    else
```

```
        free (name);

    return service;
}

//  ---------------------------------------------------------------
//  当服务从broker->services中移除时销毁该服务对象

static void
s_service_destroy (void *argument)
{
    service_t *service = (service_t *) argument;
    //  销毁请求队列中的所有项目
    while (zlist_size (service->requests)) {
        zmsg_t *msg = zlist_pop (service->requests);
        zmsg_destroy (&msg);
    }
    zlist_destroy (&service->requests);
    zlist_destroy (&service->waiting);
    free (service->name);
    free (service);
}

//  ---------------------------------------------------------------
//  可能时，分发请求给等待中的worker

static void
s_service_dispatch (broker_t *self, service_t *service, zmsg_t *msg
{
    assert (service);
    if (msg)                    //  将消息加入队列
        zlist_append (service->requests, msg);

    s_broker_purge_workers (self);
    while (zlist_size (service->waiting)
        && zlist_size (service->requests))
    {
        worker_t *worker = zlist_pop (service->waiting);
        zlist_remove (self->waiting, worker);
        zmsg_t *msg = zlist_pop (service->requests);
        s_worker_send (self, worker, MDPW_REQUEST, NULL, msg);
        zmsg_destroy (&msg);
    }
}

//  ---------------------------------------------------------------
//  使用8/MMI协定处理内部服务

static void
s_service_internal (broker_t *self, zframe_t *service_frame, zmsg_t
{
    char *return_code;
    if (zframe_streq (service_frame, "mmi.service")) {
```

```
        char *name = zframe_strdup (zmsg_last (msg));
        service_t *service =
            (service_t *) zhash_lookup (self->services, name);
        return_code = service && service->workers? "200": "404";
        free (name);
    }
    else
        return_code = "501";

    zframe_reset (zmsg_last (msg), return_code, strlen (return_code

    //  移除并保存返回给client的信封，插入协议头信息和服务名称，并重新包装信封
    zframe_t *client = zmsg_unwrap (msg);
    zmsg_push (msg, zframe_dup (service_frame));
    zmsg_pushstr (msg, MDPC_CLIENT);
    zmsg_wrap (msg, client);
    zmsg_send (&msg, self->socket);
}

//  ---------------------------------------------------------------------
//  按需创建worker

static worker_t *
s_worker_require (broker_t *self, zframe_t *address)
{
    assert (address);

    //  self->workers使用wroker的标识为键
    char *identity = zframe_strhex (address);
    worker_t *worker =
        (worker_t *) zhash_lookup (self->workers, identity);

    if (worker == NULL) {
        worker = (worker_t *) zmalloc (sizeof (worker_t));
        worker->identity = identity;
        worker->address = zframe_dup (address);
        zhash_insert (self->workers, identity, worker);
        zhash_freefn (self->workers, identity, s_worker_destroy);
        if (self->verbose)
            zclock_log ("I: 正在注册新的worker: %s", identity);
    }
    else
        free (identity);
    return worker;
}

//  ---------------------------------------------------------------------
//  从所有数据结构中删除wroker，并销毁worker对象

static void
s_worker_delete (broker_t *self, worker_t *worker, int disconnect)
{
    assert (worker);
```

```
    if (disconnect)
        s_worker_send (self, worker, MDPW_DISCONNECT, NULL, NULL);

    if (worker->service) {
        zlist_remove (worker->service->waiting, worker);
        worker->service->workers--;
    }
    zlist_remove (self->waiting, worker);
    //  以下方法间接调用了s_worker_destroy()方法
    zhash_delete (self->workers, worker->identity);
}

//  ---------------------------------------------------------------------
//  当worker从broker->workers中移除时，销毁worker对象

static void
s_worker_destroy (void *argument)
{
    worker_t *worker = (worker_t *) argument;
    zframe_destroy (&worker->address);
    free (worker->identity);
    free (worker);
}

//  ---------------------------------------------------------------------
//  处理worker发送来的消息

static void
s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);     //  消息中至少包含命令帧

    zframe_t *command = zmsg_pop (msg);
    char *identity = zframe_strhex (sender);
    int worker_ready = (zhash_lookup (self->workers, identity) != N
    free (identity);
    worker_t *worker = s_worker_require (self, sender);

    if (zframe_streq (command, MDPW_READY)) {
        //  若worker队列中已有该worker，但仍收到了它的“已就绪”消息，则删除
        if (worker_ready)
            s_worker_delete (self, worker, 1);
        else
        if (zframe_size (sender) >= 4  //  服务名称为保留的服务
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (self, worker, 1);
        else {
            //  将worker对应到服务，并置为空闲状态
            zframe_t *service_frame = zmsg_pop (msg);
            worker->service = s_service_require (self, service_fran
            worker->service->workers++;
            s_worker_waiting (self, worker);
            zframe_destroy (&service_frame);
```

```
            }
        }
        else
        if (zframe_streq (command, MDPW_REPLY)) {
            if (worker_ready) {
                //  移除并保存返回给client的信封，插入协议头信息和服务名称，并重
                zframe_t *client = zmsg_unwrap (msg);
                zmsg_pushstr (msg, worker->service->name);
                zmsg_pushstr (msg, MDPC_CLIENT);
                zmsg_wrap (msg, client);
                zmsg_send (&msg, self->socket);
                s_worker_waiting (self, worker);
            }
            else
                s_worker_delete (self, worker, 1);
        }
        else
        if (zframe_streq (command, MDPW_HEARTBEAT)) {
            if (worker_ready)
                worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
            else
                s_worker_delete (self, worker, 1);
        }
        else
        if (zframe_streq (command, MDPW_DISCONNECT))
            s_worker_delete (self, worker, 0);
        else {
            zclock_log ("E: 非法消息");
            zmsg_dump (msg);
        }
        free (command);
        zmsg_destroy (&msg);
    }

    //  ---------------------------------------------------------------
    //  发送消息给worker
    //  如果指针指向了一条消息，发送它，但不销毁它，因为这是调用者的工作

    static void
    s_worker_send (broker_t *self, worker_t *worker, char *command,
                   char *option, zmsg_t *msg)
    {
        msg = msg? zmsg_dup (msg): zmsg_new ();

        //  将协议信封压入消息顶部
        if (option)
            zmsg_pushstr (msg, option);
        zmsg_pushstr (msg, command);
        zmsg_pushstr (msg, MDPW_WORKER);

        //  在消息顶部插入路由帧
        zmsg_wrap (msg, zframe_dup (worker->address));
```

```
    if (self->verbose) {
        zclock_log ("I: 正在发送消息给worker %s",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->socket);
}

//  ---------------------------------------------------------------------
//  正在等待的worker

static void
s_worker_waiting (broker_t *self, worker_t *worker)
{
    //  将worker加入代理和服务的等待队列
    zlist_append (self->waiting, worker);
    zlist_append (worker->service->waiting, worker);
    worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
    s_service_dispatch (self, worker->service, NULL);
}

//  ---------------------------------------------------------------------
//  处理client发来的请求

static void
s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);     //  服务名称 + 请求内容

    zframe_t *service_frame = zmsg_pop (msg);
    service_t *service = s_service_require (self, service_frame);

    //  为应答内容设置请求方的地址
    zmsg_wrap (msg, zframe_dup (sender));
    if (zframe_size (service_frame) >= 4
    &&  memcmp (zframe_data (service_frame), "mmi.", 4) == 0)
        s_service_internal (self, service_frame, msg);
    else
        s_service_dispatch (self, service, msg);
    zframe_destroy (&service_frame);
}
```

这个例子应该是我们见过最复杂的一个示例了，大约有500行代码。编写这段代码并让其变的健壮，大约花费了两天的时间。但是，这也仅仅是一个完整的面向服务代理的一部分。

几点说明：

- 管家模式协议要求我们在一个套接字中同时处理client和worker，这一点对部署和管理代理很有益处：它只会在一个ZMQ端点上收发请求，而不是两个。

- 代理很好地实现了MDP/0.1协议中规范的内容，包括当代理发送非法命令和心跳时断开的机制。
- 可以将这段代码扩充为多线程，每个线程管理一个套接字、一组client和worker。这种做法在大型架构的拆分中显得很有趣。C语言代码已经是这样的格式了，因此很容易实现。
- 还可以将这段代码扩充为主备模式、双在线模式，进一步提高可靠性。因为从本质上来说，代理是无状态的，只是保存了服务的存在与否，因此client和worker可以自行选择除此之外的代理来进行通信。
- 示例代码中心跳的间隔为5秒，主要是为了减少调试时的输出。现实中的值应该设得低一些，但是，重试的过程应该设置得稍长一些，让服务有足够的时间启动，如10秒钟。

异步管家模式

上文那种实现管家模式的方法比较简单，client还是简单海盗模式中的，仅仅是用API重写了一下。我在测试机上运行了程序，处理10万条请求大约需要14秒的时间，这和代码也有一些关系，因为复制消息帧的时间浪费了CPU处理时间。但真正的问题在于，我们总是逐个循环进行处理（round-trip），即发送-接收-发送-接收……ZMQ内部禁用了TCP发包优化算法（Nagle's algorithm），但逐个处理循环还是比较浪费。

理论归理论，还是需要由实践来检验。我们用一个简单的测试程序来看看逐个处理循环是否真的耗时。这个测试程序会发送一组消息，第一次它发一条收一条，第二次则一起发送再一起接收。两次结果应该是一样的，但速度截然不同。

tripping: Round-trip demonstrator in C

```
//
//  Round-trip 模拟
//
//  本示例程序使用多线程的方式启动client、worker、以及代理，
//  当client处理完毕时会发送信号给主程序。
//
#include "czmq.h"

static void
client_task (void *args, zctx_t *ctx, void *pipe)
{
    void *client = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (client, ZMQ_IDENTITY, "C", 1);
    zsocket_connect (client, "tcp://localhost:5555");

    printf ("开始测试...\n");
    zclock_sleep (100);

    int requests;
    int64_t start;
```

```
    printf ("同步 round-trip 测试...\n");
    start = zclock_time ();
    for (requests = 0; requests < 10000; requests++) {
        zstr_send (client, "hello");
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d 次/秒\n",
        (1000 * 10000) / (int) (zclock_time () - start));

    printf ("异步 round-trip 测试...\n");
    start = zclock_time ();
    for (requests = 0; requests < 100000; requests++)
        zstr_send (client, "hello");
    for (requests = 0; requests < 100000; requests++) {
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d 次/秒\n",
        (1000 * 100000) / (int) (zclock_time () - start));

    zstr_send (pipe, "完成");
}

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "W", 1);
    zsocket_connect (worker, "tcp://localhost:5556");

    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

static void *
broker_task (void *args)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");
    zsocket_bind (backend,  "tcp://*:5556");

    //  初始化轮询对象
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
```

```
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    while (1) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1)
            break;              //  中断
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "W");
            zmsg_send (&msg, backend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "C");
            zmsg_send (&msg, frontend);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    //  创建线程
    zctx_t *ctx = zctx_new ();
    void *client = zthread_fork (ctx, client_task, NULL);
    zthread_new (ctx, worker_task, NULL);
    zthread_new (ctx, broker_task, NULL);

    //  等待client端管道的信号
    char *signal = zstr_recv (client);
    free (signal);

    zctx_destroy (&ctx);
    return 0;
}
```

在我的开发环境中运行结果如下：

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

需要注意的是client在运行开始会暂停一段时间，这是因为在向ROUTER套接字发送消息时，若指定标识的套接字没有连接，那么ROUTER会直接丢弃该消息。这个示例中我们没有使用LRU算法，所以当worker连接速度稍慢时就有可能丢失数据，影响测试结果。

我们可以看到，逐个处理循环比异步处理要慢将近20倍，让我们把它应用到管家模式中去。

首先，让我们修改client的API，添加独立的发送和接收方法：

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
int      mdcli_send    (mdcli_t *self, char *service, zmsg_t **requ
zmsg_t  *mdcli_recv    (mdcli_t *self);
```

然后花很短的时间就能将同步的client API改造成异步的API：

mdcliapi2: Majordomo asynchronous client API in C

```
/*  =================================================================
    mdcliapi2.c

    Majordomo Protocol Client API (async version)
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    -----------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =================================================================
*/

#include "mdcliapi2.h"

//  类结构
```

```
//  使用成员函数访问属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  连接至代理的套接字
    int verbose;                //  在标准输出打印运行状态
    int timeout;                //  请求超时时间
};


//  ---------------------------------------------------------------------
//  连接或重连代理

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: 正在连接代理 %s...", self->broker);
}


//  ---------------------------------------------------------------------
//  构造函数

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  毫秒

    s_mdcli_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
```

```
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  设置请求超时时间

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}


//  ---------------------------------------------------------------------
//  发送请求给代理
//  取得请求消息的所有权，发送后销毁

int
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  在消息顶部加入协议规定的帧
    //  Frame 0: empty (模拟REQ套接字的行为)
    //  Frame 1: "MDPCxy" (6个字节, MDP/Client x.y)
    //  Frame 2: Service name (看打印字符串)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    zmsg_pushstr (request, "");
    if (self->verbose) {
        zclock_log ("I: 发送请求给 '%s' 服务:", service);
        zmsg_dump (request);
    }
    zmsg_send (&request, self->client);
    return 0;
}


//  ---------------------------------------------------------------------
//  获取应答消息，若无则返回NULL；
//  该函数不会尝试从代理的崩溃中恢复，
//  因为我们没有记录那些未收到应答的请求，所以也无法重发。

zmsg_t *
mdcli_recv (mdcli_t *self)
```

```
{
    assert (self);

    //  轮询套接字以获取应答
    zmq_pollitem_t items [] = { { self->client, 0, ZMQ_POLLIN, 0 }
    int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
    if (rc == -1)
        return NULL;            //  中断

    //  收到应答后进行处理
    if (items [0].revents & ZMQ_POLLIN) {
        zmsg_t *msg = zmsg_recv (self->client);
        if (self->verbose) {
            zclock_log ("I: received reply:");
            zmsg_dump (msg);
        }
        //  不要处理错误，直接报出
        assert (zmsg_size (msg) >= 4);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPC_CLIENT));
        zframe_destroy (&header);

        zframe_t *service = zmsg_pop (msg);
        zframe_destroy (&service);

        return msg;     //  Success
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，正在中止client...\n");
    else
    if (self->verbose)
        zclock_log ("W: 严重错误，放弃请求");

    return NULL;
}
```

下面是对应的测试代码：

mdclient2: Majordomo client application in C

```
//
//  异步管家模式 - client示例程序
//  使用mdcli API隐藏MDP协议的具体实现
//
//  直接编译源码，而不创建类库
#include "mdcliapi2.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        mdcli_send (session, "echo", &request);
    }
    for (count = 0; count < 100000; count++) {
        zmsg_t *reply = mdcli_recv (session);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  使用Ctrl-C中断
    }
    printf ("收到 %d 个应答\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

代理和worker的代码没有变，因为我们并没有改变MDP协议。经过对client的改造，我们可以明显看到速度的提升。如以下是同步状况下处理10万条请求的时间：

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

以下是异步请求的情况：

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

让我们建立10个worker，看看效果如何：

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

由于worker获得消息需要通过LRU队列机制，所以并不能做到完全的异步。但是，worker越多其效果也会越好。在我的测试机上，当worker的数量达到8个时，速度就不再提升了——四核处理器只能做这么多。但是，我们仍然获得了近四倍的速度提升，而改造过程只有几分钟而已。此外，代理其实还没有进行优化，它仍会复制消息，而没有实现零拷贝。不过，我们已经做到每秒处理2.5万次请求-应答，已经很不错了。

当然，异步的管家模式也并不完美，有一个显著的缺点：它无法从代理的崩溃中恢复。可以看到mdcliapi2的代码中并没有恢复连接的代码，重新连接需要有以下几点作为前提：

- 每个请求都做了编号，每次应答也含有相应的编号，这就需要修改协议，明确定义；
- client的API需要保留并跟踪所有已发送、但仍未收到应答的请求；
- 如果代理发生崩溃，client会重发所有消息。

可以看到，高可靠性往往和复杂度成正比，值得在管家模式中应用这一机制吗？这就要看应用场景了。如果是一个名称查询服务，每次会话会调用一次，那不需要应用这一机制；如果是一个位于前端的网页服务，有数千个客户端相连，那可能就需要了。

服务查询

现在，我们已经有了一个面向服务的代理了，但是我们无法得知代理是否提供了某项特定服务。如果请求失败，那当然就表示该项服务目前不可用，但具体原因是什么呢？所以，如果能够询问代理“echo服务正在运行吗？”，那将会很有用处。最明显的方法是在MDP/Client协议中添加一种命令，客户端可以询问代理某项服务是否可用。但是，MDP/Client最大的优点在于简单，如果添加了服务查询的功能就太过复杂了。

另一种方案是学电子邮件的处理方式，将失败的请求重新返回。但是这同样会增加复杂度，因为我们需要鉴别收到的消息是一个应答还是被退回的请求。

让我们用之前的方式，在MDP的基础上建立新的机制，而不是改变它。服务定位本身也是一项服务，我们还可以提供类似于“禁用某服务”、“提供服务数据”等其他服务。我们需要的是一个能够扩展协议但又不会影响协议本身的机制。

这样就诞生了一个小巧的RFC - MMI（管家接口）的应用层，建立在MDP协议之上：http://rfc.zeromq.org/spec:8 。我们在代理中其实已经加以实现了，不知你是否已经注意到。下面的代码演示了如何使用这项服务查询功能：

mmiecho: Service discovery over Majordomo in C

```
//
//  MMI echo 服务查询示例程序
//

//  让我们直接编译，不生成类库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  我们需要查询的服务名称
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    //  发送给“服务查询”服务的消息
    zmsg_t *reply = mdcli_send (session, "mmi.service", &request);

    if (reply) {
        char *reply_code = zframe_strdup (zmsg_first (reply));
        printf ("查询 echo 服务的结果: %s\n", reply_code);
        free (reply_code);
        zmsg_destroy (&reply);
    }
    else
        printf ("E: 代理无响应，请确认它正在工作\n");

    mdcli_destroy (&session);
    return 0;
}
```

代理在运行时会检查请求的服务名称，自行处理那些mmi.开头的服务，而不转发给worker。你可以在不开启worker的情况下运行以上代码，可以看到程序是报告200还是404。MMI在示例程序代理中的实现是很简单的，比如，当某个worker消亡时，该服务仍然标记为可用。实践中，代理应该在一定间隔后清除那些没有worker的服务。

幂等服务

幂等是指能够安全地重复执行某项操作。如，看钟是幂等的，但借钱给别人老婆就不是了。有些客户端至服务端的通信是幂等的，但有些则不是。幂等的通信示例有：

- 无状态的任务分配，即管道模式中服务端是无状态的worker，它的处理结果是根据客户端的请求状态生成的，因此可以重复处理相同的请求；
- 命名服务中将逻辑地址转化成实际绑定或连接的端点，可以重复查询多次，因此也是幂等的。

非幂等的通信示例有：

- 日志服务，我们不会希望相同的日志内容被记录多次；
- 任何会对下游节点有影响的服务，如该服务会向下游节点发送信息，若收到相同的请求，那下游节点收到的信息就是重复的；
- 当服务修改了某些共享的数据，且没有进行幂等方面的设置。如某项服务对银行账户进行了借操作（debit），这一定是非幂等的。

如果应用程序提供的服务是非幂等的，那就需要考虑它究竟是在哪个阶段崩溃的。如果程序在空闲或处理请求的过程中崩溃，那不会有什么问题。我们可以使用数据库中的事务机制来保证借贷操作是同时发生的。如果应用程序在发送请求的时候崩溃了，那就会有问题，因为对于该程序来说，它已经完成了工作。

如果在返回应答的过程中网络阻塞了，客户端会认为请求发送失败，并进行重发，这样服务端会再一次执行相同的请求。这不是我们想要的结果。

常用的解决方法是在服务端检测并拒绝重复的请求，这就需要：

- 客户端为每个请求加注唯一的标识，包括客户端标识和消息标识；
- 服务端在发送应答时使用客户端标识和消息标识作为键，保存应答内容；
- 当服务端发现收到的请求已在应答哈希表中存在，它会跳过该次请求，直接返回应答内容。

脱机可靠性（巨人模式）

当你意识到管家模式是一种非常可靠的消息代理时，你可能会想要使用磁盘做一下消息中转，从而进一步提升可靠性。这种方式虽然在很多企业级消息系统中应用，但我还是有些反对的，原因有：

- 我们可以看到，懒惰海盗模式的client可以工作得非常好，能够在多种架构中运行。唯一的问题是它会假设worker是无状态的，且提供的服务是幂等的。但这个问题我们可以通过其他方式解决，而不是添加磁盘。

- 添加磁盘会带来新的问题，需要额外的管理和维护费用。海盗模式的最大优点就是简单明了，不会崩溃。如果你还是担心硬件会出问题，可以改用点对点的通信模式，这会在本章最后一节讲到。

虽然有以上原因，但还是有一个合理的场景可以用到磁盘中转的——异步脱机网络。海盗模式有一个问题，那就是client发送请求后会一直等待应答。如果client和worker并不是长连接（可以拿电子邮箱做个类比），我们就无法在client和worker之间建立一个无状态的网络，因此需要将这种状态保存起来。

于是我们就有了巨人模式，该模式下会将消息写到磁盘中，确保不会丢失。当我们进行服务查询时，会转向巨人这一层进行。巨人是建立在管家之上的，而不是改写了MDP协议。这样做的好处是我们可以在一个特定的worker中实现这种可靠性，而不用去增加代理的逻辑。

- 实现更为简单；
  - 代理用一种语言编写，worker使用另一种语言编写；
  - 可以自由升级这种模式。

唯一的缺点是，代理和磁盘之间会有一层额外的联系，不过这也是值得的。

我们有很多方法来实现一种持久化的请求-应答架构，而目标当然是越简单越好。我能想到的最简单的方式是提供一种成为“巨人”的代理服务，它不会影响现有worker的工作，若client想要立即得到应答，它可以和代理进行通信；如果它不是那么着急，那就可以和巨人通信：“嗨，巨人，麻烦帮我处理下这个请求，我去买些菜。”

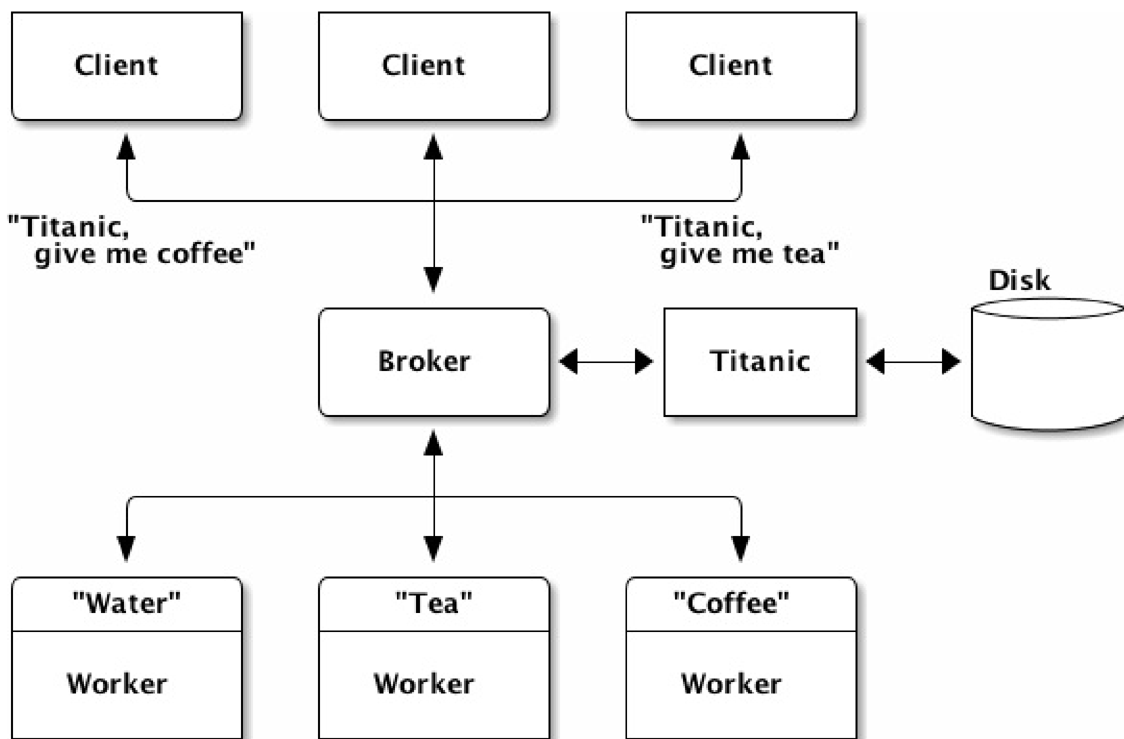


Figure 5 – Titanic Pattern

这样一来，巨人就既是worker又是client。client和巨人之间的对话一般是：

- Client: 请帮我处理这个请求。巨人：好的。
- Client: 有要给我的应答吗？巨人：有的。（或者没有）
- Client: OK，你可以释放那个请求了，工作已经完成。巨人：好的。

巨人和代理之间的对话一般是：

- 巨人：嗨，代理程序，你这里有个叫echo的服务吗？代理：恩，好像有。
- 巨人：嗨，echo服务，请帮我处理一下这个请求。Echo: 好了，这是应答。
- 巨人：谢谢！

你可以想象一些发生故障的情形，看看上述模式是否能解决？worker在处理请求的时候崩溃，巨人会不断地重新发送请求；应答在传输过程中丢失了，巨人也会重试；如果请求已经处理，但client没有得到应答，那它会再次询问巨人；如果巨人在处理请求或进行应答的时候崩溃了，客户端会进行重试；只要请求是被保存在磁盘上的，那它就不会丢失。

这个机制中，握手的过程是比较漫长的，但client可以使用异步的管家模式，一次发送多个请求，并一起等待应答。

我们需要一种方法，让client会去请求应答内容。不同的client会访问到相同的服务，且client是来去自由的，有着不同的标识。一个简单、合理、安全的解决方案是：

- 当巨人收到请求时，它会为每个请求生成唯一的编号（UUID），并将这个编号返回给client；
- client在请求应答内容时需要提供这个编号。

这样一来client就需要负责将UUID安全地保存起来，不过这就省去了验证的过程。有其他方案吗？我们可以使用持久化的套接字，即显式声明客户端的套接字标识。然而，这会造成管理上的麻烦，而且万一两个client的套接字标识相同，那会引来无穷的麻烦。

在我们开始制定一个新的协议之前，我们先思考一下client如何和巨人通信。一种方案是提供一种服务，配合三个不同的命令；另一种方案则更为简单，提供三种独立的服务：

- **titanic.request** - 保存一个请求，并返回UUID
- **titanic.reply** - 根据UUID获取应答内容
- **titanic.close** - 确认某个请求已被正确地处理

我们需要创建一个多线程的worker，正如我们之前用ZMQ进行多线程编程一样，很简单。但是，在我们开始编写代码之前，先讲巨人模式的一些定义写下来：http://rfc.zeromq.org/spec:9 。我们称之为“巨人服务协议”，或TSP。

使用TSP协议自然会让client多出额外的工作，下面是一个简单但足够健壮的client：

ticlient: Titanic client example in C

```
//
//  巨人模式client示例
//  实现 http://rfc.zeromq.org/spec:9 协议中的client端

//  让我们直接编译，不创建类库
#include "mdcliapi.c"

//  请求TSP协议下的服务
```

```
//  如果成功则返回应答（状态码：200），否则返回NULL
//
static zmsg_t *
s_service_call (mdcli_t *session, char *service, zmsg_t **request_p)
{
    zmsg_t *reply = mdcli_send (session, service, request_p);
    if (reply) {
        zframe_t *status = zmsg_pop (reply);
        if (zframe_streq (status, "200")) {
            zframe_destroy (&status);
            return reply;
        }
        else
        if (zframe_streq (status, "400")) {
            printf ("E: 客户端发生严重错误，取消请求\n");
            exit (EXIT_FAILURE);
        }
        else
        if (zframe_streq (status, "500")) {
            printf ("E: 服务端发生严重错误，取消请求\n");
            exit (EXIT_FAILURE);
        }
    }
    else
        exit (EXIT_SUCCESS);    //  中断或发生错误

    zmsg_destroy (&reply);
    return NULL;        //  请求不成功，但不返回失败原因
}

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  1. 发送echo服务的请求给巨人
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = s_service_call (
        session, "titanic.request", &request);

    zframe_t *uuid = NULL;
    if (reply) {
        uuid = zmsg_pop (reply);
        zmsg_destroy (&reply);
        zframe_print (uuid, "I: request UUID ");
    }

    //  2. 等待应答
    while (!zctx_interrupted) {
        zclock_sleep (100);
        request = zmsg_new ();
```

```
            zmsg_add (request, zframe_dup (uuid));
            zmsg_t *reply = s_service_call (
                session, "titanic.reply", &request);

            if (reply) {
                char *reply_string = zframe_strdup (zmsg_last (reply));
                printf ("Reply: %s\n", reply_string);
                free (reply_string);
                zmsg_destroy (&reply);

                //  3. 关闭请求
                request = zmsg_new ();
                zmsg_add (request, zframe_dup (uuid));
                reply = s_service_call (session, "titanic.close", &requ
                zmsg_destroy (&reply);
                break;
            }
            else {
                printf ("I: 尚未收到应答，准备稍后重试...\n");
                zclock_sleep (5000);     //  5秒后重试
            }
        }
        zframe_destroy (&uuid);
        mdcli_destroy (&session);
        return 0;
    }
```

当然，上面的代码可以整合到一个框架中，程序员不需要了解其中的细节。如果我有时间的话，我会尝试写一个这样的API的，让应用程序又变回短短的几行。这种理念和MDP中的一致：不要做重复的事。

下面是巨人的实现。这个服务端会使用三个线程来处理三种服务。它使用最原始的持久化方法来保存请求：为每个请求创建一个磁盘文件。虽然简单，但也挺恐怖的。比较复杂的部分是，巨人会维护一个队列来保存这些请求，从而避免重复地扫描目录。

titanic: Titanic broker example in C

```
//
//  巨人模式 - 服务
//
//  实现 http://rfc.zeromq.org/spec:9 协议的服务端

//  让我们直接编译，不创建类库
#include "mdwrkapi.c"
#include "mdcliapi.c"

#include "zfile.h"
#include <uuid/uuid.h>
```

```
//  返回一个可打印的唯一编号（UUID）
//  调用者负责释放UUID字符串的内存

static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15
    }
    return uuidstr;
}

//  根据UUID生成用于保存请求内容的文件名，并返回

#define TITANIC_DIR ".titanic"

static char *
s_request_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.req", uuid);
    return filename;
}

//  根据UUID生成用于保存应答内容的文件名，并返回

static char *
s_reply_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.rep", uuid);
    return filename;
}


//  ---------------------------------------------------------------------
//  巨人模式 - 请求服务

static void
titanic_request (void *args, zctx_t *ctx, void *pipe)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.request", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        //  若应答非空则发送，再从代理处获得新的请求
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
```

```
                break;      //  中断并退出

            //  确保消息目录是存在的
            file_mkdir (TITANIC_DIR);

            //  生成UUID，并将消息保存至磁盘
            char *uuid = s_generate_uuid ();
            char *filename = s_request_filename (uuid);
            FILE *file = fopen (filename, "w");
            assert (file);
            zmsg_save (request, file);
            fclose (file);
            free (filename);
            zmsg_destroy (&request);

            //  将UUID加入队列
            reply = zmsg_new ();
            zmsg_addstr (reply, uuid);
            zmsg_send (&reply, pipe);

            //  将UUID返回给客户端
            //  将由循环顶部的mdwrk_recv()函数完成
            reply = zmsg_new ();
            zmsg_addstr (reply, "200");
            zmsg_addstr (reply, uuid);
            free (uuid);
        }
        mdwrk_destroy (&worker);
}


//  ---------------------------------------------------------------------
//  巨人模式 - 应答服务

static void *
titanic_reply (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.reply", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断并退出

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        if (file_exists (rep_filename)) {
            FILE *file = fopen (rep_filename, "r");
            assert (file);
            reply = zmsg_load (file);
```

```
            zmsg_pushstr (reply, "200");
            fclose (file);
        }
        else {
            reply = zmsg_new ();
            if (file_exists (req_filename))
                zmsg_pushstr (reply, "300"); //挂起
            else
                zmsg_pushstr (reply, "400"); //未知
        }
        zmsg_destroy (&request);
        free (uuid);
        free (req_filename);
        free (rep_filename);
    }
    mdwrk_destroy (&worker);
    return 0;
}


//  ---------------------------------------------------------------------
//  巨人模式 - 关闭请求

static void *
titanic_close (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.close", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断并退出

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        file_delete (req_filename);
        file_delete (rep_filename);
        free (uuid);
        free (req_filename);
        free (rep_filename);

        zmsg_destroy (&request);
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
    }
    mdwrk_destroy (&worker);
    return 0;
}

//  处理某个请求，成功则返回1
```

```
static int
s_service_success (mdcli_t *client, char *uuid)
{
    //  读取请求内容，第一帧为服务名称
    char *filename = s_request_filename (uuid);
    FILE *file = fopen (filename, "r");
    free (filename);

    //  如果client已经关闭了该请求，则返回1
    if (!file)
        return 1;

    zmsg_t *request = zmsg_load (file);
    fclose (file);
    zframe_t *service = zmsg_pop (request);
    char *service_name = zframe_strdup (service);

    //  使用MMI协议检查服务是否可用
    zmsg_t *mmi_request = zmsg_new ();
    zmsg_add (mmi_request, service);
    zmsg_t *mmi_reply = mdcli_send (client, "mmi.service", &mmi_rec
    int service_ok = (mmi_reply
        && zframe_streq (zmsg_first (mmi_reply), "200"));
    zmsg_destroy (&mmi_reply);

    if (service_ok) {
        zmsg_t *reply = mdcli_send (client, service_name, &request)
        if (reply) {
            filename = s_reply_filename (uuid);
            FILE *file = fopen (filename, "w");
            assert (file);
            zmsg_save (reply, file);
            fclose (file);
            free (filename);
            return 1;
        }
        zmsg_destroy (&reply);
    }
    else
        zmsg_destroy (&request);

    free (service_name);
    return 0;
}


int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    zctx_t *ctx = zctx_new ();

    //  创建MDP客户端会话
```

```
    mdcli_t *client = mdcli_new ("tcp://localhost:5555", verbose);
    mdcli_set_timeout (client, 1000);  //  1 秒
    mdcli_set_retries (client, 1);     //  只尝试一次

    void *request_pipe = zthread_fork (ctx, titanic_request, NULL);
    zthread_new (ctx, titanic_reply, NULL);
    zthread_new (ctx, titanic_close, NULL);

    //  主循环
    while (TRUE) {
        //  如果没有活动，我们将每秒循环一次
        zmq_pollitem_t items [] = { { request_pipe, 0, ZMQ_POLLIN,
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断
        if (items [0].revents & ZMQ_POLLIN) {
            //  确保消息目录是存在的
            file_mkdir (TITANIC_DIR);

            //  将UUID添加到队列中，使用“-”号标识等待中的请求
            zmsg_t *msg = zmsg_recv (request_pipe);
            if (!msg)
                break;          //  中断
            FILE *file = fopen (TITANIC_DIR "/queue", "a");
            char *uuid = zmsg_popstr (msg);
            fprintf (file, "-%s\n", uuid);
            fclose (file);
            free (uuid);
            zmsg_destroy (&msg);
        }
        //  分派
        //
        char entry [] = "?.......:.......:.......:.......:";
        FILE *file = fopen (TITANIC_DIR "/queue", "r+");
        while (file && fread (entry, 33, 1, file) == 1) {
            //  处理UUID前缀为“-”的请求
            if (entry [0] == '-') {
                if (verbose)
                    printf ("I: 开始处理请求 %s\n", entry + 1);
                if (s_service_success (client, entry + 1)) {
                    //  标记为已处理
                    fseek (file, -33, SEEK_CUR);
                    fwrite ("+", 1, 1, file);
                    fseek (file, 32, SEEK_CUR);
                }
            }
            //  跳过最后一行
            if (fgetc (file) == '\r')
                fgetc (file);
            if (zctx_interrupted)
                break;
        }
        if (file)
```

```
            fclose (file);
    }
    mdcli_destroy (&client);
    return 0;
}
```

测试时，打开mdbroker和titanic，再运行ticlient，然后开启任意个mdworker，就可以看到client获得了应答。

几点说明：

- 我们使用MMI协议去向代理询问某项服务是否可用，这一点和MDP中的逻辑一致；
- 我们使用inproc（进程内）协议建立主循环和titanic.request服务间的联系，保存新的请求信息。这样可以避免主循环不断扫描磁盘目录，读取所有请求文件，并按照时间日期排序。

这个示例程序不应关注它的性能（一定会非常糟糕，虽然我没有测试过），而是应该看到它是如何提供一种可靠的通信模式的。你可以测试一下，打开代理、巨人、worker和client，使用-v参数显示跟踪信息，然后随意地开关代理、巨人、或worker（client不能关闭），可以看到所有的请求都能获得应答。

如果你想在真实环境中使用巨人模式，你肯定会问怎样才能让速度快起来。以下是我的做法：

- 使用一个磁盘文件保存所有数据。操作系统处理大文件的效率要比处理许多小文件来的高。
- 使用一种循环的机制来组织该磁盘文件的结构，这样新的请求可以被连续地写入这个文件。单个线程在全速写入磁盘时的效率是比较高的。
- 将索引保存在内存中，可以在启动程序时重建这个索引。这样做可以节省磁盘缓存，让索引安全地保存在磁盘上。你需要用到fsync的机制来保存每一条数据；或者可以等待几毫秒，如果不怕丢失上千条数据的话。
- 如果条件允许，应选择使用固态硬盘；
- 提前分配该磁盘文件的空间，或者将每次分配的空间调大一些，这样可以避免磁盘碎片的产生，并保证读写是连续的。

另外，我不建议将消息保存在数据库中，甚至不建议交给那些所谓的高速键值缓存，它们比起一个磁盘文件要来得昂贵。

如果你想让巨人模式变得更为可靠，你可以将请求复制到另一台服务器上，这样就不需要担心主程序遭到核武器袭击了。

如果你想让巨人模式变得更为快速，但可以牺牲一些可靠性，那你可以将请求和应答都保存在内存中。这样做可以让该服务作为脱机网络运行，不过若巨人服务本身崩溃了，我也无能为力。

高可靠对称节点（双子星模式）

概览

双子星模式是一对具有主从机制的高可靠节点。任一时间，某个节点会充当主机，接收所有客户端的请求；另一个则作为一种备机存在。两个节点会互相监控对方，当主机从网络中消失时，备机会替代主机的位置。

双子星模式由Pieter Hintjens和Martin Sustrik设计，应用在iMatix的OpenAMQ服务器中。它的设计理念是：

- 提供一种简明的高可靠性解决方案；
- 易于理解和使用；
- 能够进行可靠的故障切换。

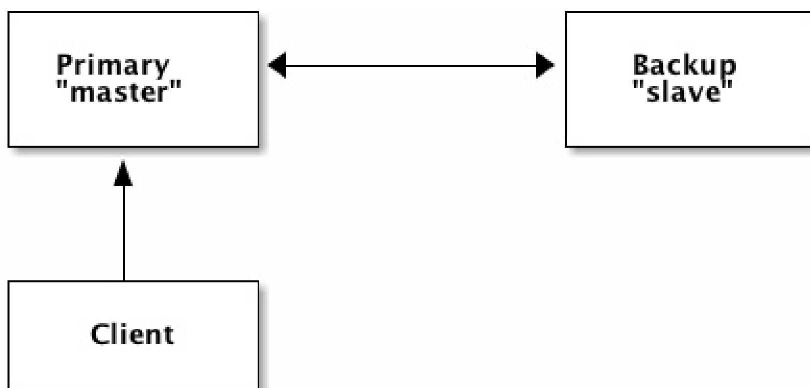


Figure 6 – –High availability pair, normal operation

假设我们有一组双子星模式的服务器，以下是可能发生的故障：

1. 主机发生硬件故障（断电、失火等），应用程序发送后立刻使用备机进行连接；
2. 主机的网络环境发生故障，可能某个路由器被雷击了，立刻使用备机；
3. 主机上的服务被维护人员误杀，无法自动恢复。

恢复步骤如下：

1. 维护人员排查主机故障；
2. 将备机关闭，造成短时间的服务不可用；
3. 待应用程序都连接到主机后，维护人员重启备机。

恢复过程是人工进行的，惨痛的经验告诉我们自动恢复是很可怕的：

- 故障的发生会造成10-30秒之间的服务暂停，如果这是一个真正的突发状况，那最好还是让主机暂停服务的好，因为立刻重启服务可能造成另一个10-30秒的暂停，不如让用户停止使用。

- 当有紧急状况发生时，可以在修复的过程中记录故障发生原因，而不是让系统自动恢复，管理员因此无法用其经验抵御下一次突发状况。

- 最后，如果自动恢复确实成功了，管理员将无从得知故障的发生原因，因而无法进行分析。

双子星模式的故障恢复过程是：在修复了主机的问题后，将备机做关闭处理，稍后再重新开启：

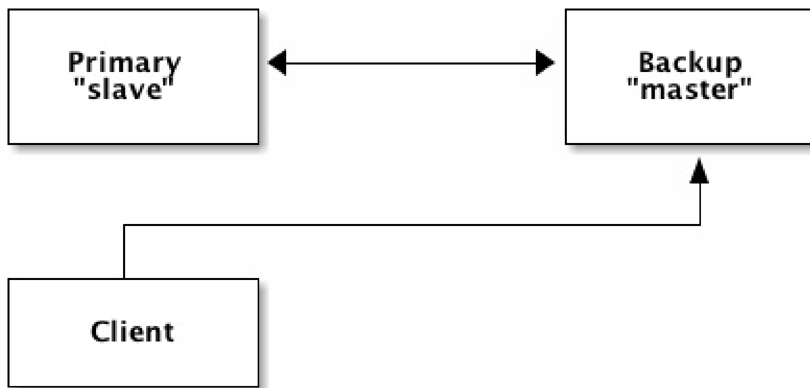


Figure 7 — —High availability pair, during failover

双子星模式的关闭过程有两种：

1. 先关闭备机，等待一段时间后再关闭主机；
2. 同时关闭主机和备机，间隔时间不超过几秒。

关闭时，间隔时间要比故障切换时间短，否则会导致应用程序失去连接、重新连接、并再次失去连接，导致用户投诉。

详细要求

双子星模式可以非常简单，但能工作得很出色。事实上，这里的实现方法已经历经三个版本了，之前的版本都过于复杂，想要做太多的事情，因而被我们抛弃。我们需要的只是最基本的功能，能够提供易理解、易开发、高可靠的解决方法就可以了。

以下是该架构的详细需求：

- 需要用到双子星模式的故障是：系统遭受灾难性的打击，如硬件崩溃、火灾、意外等。对于其他常规的服务器故障，可以用更简单的方法。
- 故障恢复时间应该在60秒以内，理想情况下应该在10秒以内；
- 故障恢复(failover)应该是自动完成的，而系统还原(recover)则是由人工完成的。我们希望应用程序能够在发生故障时自动从主机切换到备机，但不希望在问题解决之前自动切换回主机，因为这很有可能让主机再次崩溃。
- 程序的逻辑应该尽量简单，易于使用，最好能封装在API中；
- 需要提供一个明确的指示，哪台主机正在提供服务，以避免“精神分裂”的症状，即两台服务器都认为自己是主机；
- 两台服务器的启动顺序不应该有限制；
- 启动或关闭主从机时不需要更改客户端的配置，但有可能会中断连接；
- 管理员需要能够同时监控两台机器；
- 两台机器之间必须有专用的高速网络连接，必须能使用特定IP进行路由。

我们做如下架假设：

- 单台备机能够提供足够的保障，不需要再进行其他备份机制；

- 主从机应该都能够提供完整的服务，承载相同的压力，不需要进行负载均衡；
- 预算中允许有这样一台长时间闲置的备机。

双子星模式不会用到：

- 多台备机，或在主从机之间进行负载均衡。该模式中的备机将一直处于空闲状态，只有主机发生问题时才会工作；
- 处理持久化的消息或事务。我们假设所连接的网络是不可靠的（或不可信的）。
- 自动搜索网络。双子星模式是手工配置的，他们知道对方的存在，应用程序则知道双子星的存在。
- 主从机之间状态的同步。所有服务端的状态必须能由应用程序进行重建。

以下是双子星模式中的几个术语：

- 主机 - 通常情况下作为master的机器；
- 备机 - 通常情况下作为slave的机器，只有当主机从网络中消失时，备机才会切换成master状态，接收所有的应用程序请求；
- **master** - 双子星模式中接收应用程序请求的机器；同一时刻只有一台master；
- **slave** - 当master消失时用以顶替的机器。

配置双子星模式的步骤：

1. 让主机知道备机的位置；
2. 让备机知道主机的位置；
3. 调整故障恢复时间，两台机器的配置必须相同。

比较重要的配置是应让两台机器间隔多久检查一次对方的状态，以及多长时间后采取行动。在我们的示例中，故障恢复时间设置为2000毫秒，超过这个时间备机就会代替主机的位置。但若你将主机的服务包裹在一个shell脚本中进行重启，就需要延长这个时间，否则备机可能在主机恢复连接的过程中转换成master。

要让客户端应用程序和双子星模式配合，你需要做的是：

1. 知道两台服务器的地址；
2. 尝试连接主机，若失败则连接备机；
3. 检测失效的连接，一般使用心跳机制；
4. 尝试重连主机，然后再连接备机，其间的间隔应比服务器故障恢复时间长；
5. 重建服务器端需要的所有状态数据；
6. 如果要保证可靠性，应重发故障期间的消息。

这不是件容易的事，所以我们一般会将其封装成一个API，供程序员使用。

双子星模式的主要限制有：

- 服务端进程不能涉及到一个以上的双子星对称节点；
- 主机只能有一个备机；
- 当备机处于slave状态时，它不会处理任何请求；
- 备机必须能够承受所有的应用程序请求；
- 故障恢复时间不能在运行时调整；
- 客户端应用程序需要做一些重连的工作。

防止精神分裂

“精神分裂”症状指的是一个集群中的不同部分同时认为自己是master，从而停止对对方的检测。双子星模式中的算法会降低这种症状的发生几率：主备机在决定自己是否为master时会检测自身是否收到了应用程序的请求，以及对方是否已经从网络中消失。

但在某些情况下，双子星模式也会发生精神分裂。比如说，主备机被配置在两幢大楼里，每幢大楼的局域网中又分布了一些应用程序。这样，当两幢大楼的网络通信被阻断，双子星模式的主备机就会分别在两幢大楼里接受和处理请求。

为了防止精神分裂，我们必须让主备机使用专用的网络进行连接，最简单的方法当然是用一根双绞线将他们相连。

我们不能将双子星部署在两个不同的岛屿上，为各自岛屿的应用程序服务。这种情况下，我们会使用诸如联邦模式的机制进行可靠性设计。

最好但最夸张的做法是，将两台机器之间的连接和应用程序的连接完全隔离开来，甚至是使用不同的网卡，而不仅仅是不同的端口。这样做也是为了日后排查错误时更为明确。

实现双子星模式

闲话少说，下面是双子星模式的服务端代码：

bstarsrv: Binary Star server in C

```
//
//  双子星模式 - 服务端
//
#include "czmq.h"

//  发送状态信息的间隔时间
//  如果对方在两次心跳过后都没有应答，则视为断开
#define HEARTBEAT 1000          //  In msecs

//  服务器状态枚举
typedef enum {
    STATE_PRIMARY = 1,          //  主机，等待同伴连接
    STATE_BACKUP = 2,           //  备机，等待同伴连接
    STATE_ACTIVE = 3,           //  激活态，处理应用程序请求
    STATE_PASSIVE = 4           //  被动态，不接收请求
} state_t;

//  对话节点事件
typedef enum {
    PEER_PRIMARY = 1,           //  主机
    PEER_BACKUP = 2,            //  备机
    PEER_ACTIVE = 3,            //  激活态
    PEER_PASSIVE = 4,           //  被动态
    CLIENT_REQUEST = 5          //  客户端请求
```

```
} event_t;

//  有限状态机
typedef struct {
    state_t state;              //  当前状态
    event_t event;              //  当前事件
    int64_t peer_expiry;        //  判定节点死亡的时限
} bstar_t;


//  执行有限状态机（将事件绑定至状态）；
//  发生异常时返回TRUE。

static Bool
s_state_machine (bstar_t *fsm)
{
    Bool exception = FALSE;
    //  主机等待同伴连接
    //  该状态下接收CLIENT_REQUEST事件
    if (fsm->state == STATE_PRIMARY) {
        if (fsm->event == PEER_BACKUP) {
            printf ("I: 已连接至备机（slave），可以作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: 已连接至备机（master），可以作为slave运行。\n");
            fsm->state = STATE_PASSIVE;
        }
    }
    else
    //  备机等待同伴连接
    //  该状态下拒绝CLIENT_REQUEST事件
    if (fsm->state == STATE_BACKUP) {
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: 已连接至主机（master），可以作为slave运行。\n");
            fsm->state = STATE_PASSIVE;
        }
        else
        if (fsm->event == CLIENT_REQUEST)
            exception = TRUE;
    }
    else
    //  服务器处于激活态
    //  该状态下接受CLIENT_REQUEST事件
    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            //  若出现两台master，则抛出异常
            printf ("E: 严重错误：双master。正在退出。\n");
            exception = TRUE;
        }
    }
    else
```

```
    //  服务器处于被动态
    //  若同伴已死，CLIENT_REQUEST事件将触发故障恢复
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            printf ("I: 主机（slave）正在重启，可作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            printf ("I: 备机（slave）正在重启，可作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
            //  若出现两台slave，集群将无响应
            printf ("E: 严重错误：双slave。正在退出\n");
            exception = TRUE;
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            //  若心跳超时，同伴将成为master；
            //  此行为由客户端请求触发。
            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                //  同伴已死，转为激活态。
                printf ("I: 故障恢复，可作为master运行。\n");
                fsm->state = STATE_ACTIVE;
            }
            else
                //  同伴还在，拒绝请求。
                exception = TRUE;
        }
    }
    return exception;
}


int main (int argc, char *argv [])
{
    //  命令行参数可以为：
    //      -p  作为主机启动, at tcp://localhost:5001
    //      -b  作为备机启动, at tcp://localhost:5002
    zctx_t *ctx = zctx_new ();
    void *statepub = zsocket_new (ctx, ZMQ_PUB);
    void *statesub = zsocket_new (ctx, ZMQ_SUB);
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    bstar_t fsm = { 0 };

    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: 主机master，等待备机（slave）连接。\n");
        zsocket_bind (frontend, "tcp://*:5001");
```

```
        zsocket_bind (statepub, "tcp://*:5003");
        zsocket_connect (statesub, "tcp://localhost:5004");
        fsm.state = STATE_PRIMARY;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: 备机slave，等待主机（master）连接。\n");
        zsocket_bind (frontend, "tcp://*:5002");
        zsocket_bind (statepub, "tcp://*:5004");
        zsocket_connect (statesub, "tcp://localhost:5003");
        fsm.state = STATE_BACKUP;
    }
    else {
        printf ("Usage: bstarsrv { -p | -b }\n");
        zctx_destroy (&ctx);
        exit (0);
    }
    //  设定下一次发送状态的时间
    int64_t send_state_at = zclock_time () + HEARTBEAT;

    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { statesub, 0, ZMQ_POLLIN, 0 }
        };
        int time_left = (int) ((send_state_at - zclock_time ()));
        if (time_left < 0)
            time_left = 0;
        int rc = zmq_poll (items, 2, time_left * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文对象被关闭

        if (items [0].revents & ZMQ_POLLIN) {
            //  收到客户端请求
            zmsg_t *msg = zmsg_recv (frontend);
            fsm.event = CLIENT_REQUEST;
            if (s_state_machine (&fsm) == FALSE)
                //  返回应答
                zmsg_send (&msg, frontend);
            else
                zmsg_destroy (&msg);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  收到状态消息，作为事件处理
            char *message = zstr_recv (statesub);
            fsm.event = atoi (message);
            free (message);
            if (s_state_machine (&fsm))
                break;          //  错误，退出。
            fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
        }
        //  定时发送状态信息
        if (zclock_time () >= send_state_at) {
```

```
                char message [2];
                sprintf (message, "%d", fsm.state);
                zstr_send (statepub, message);
                send_state_at = zclock_time () + HEARTBEAT;
            }
        }
        if (zctx_interrupted)
            printf ("W: 中断\n");

        //  关闭套接字和上下文
        zctx_destroy (&ctx);
        return 0;
    }
```

下面是客户端代码：

bstarcli: Binary Star client in C

```
//
//  双子星模式 - 客户端
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000    //  毫秒
#define SETTLE_DELAY        2000    //  超时时间

int main (void)
{
    zctx_t *ctx = zctx_new ();

    char *server [] = { "tcp://localhost:5001", "tcp://localhost:50
    uint server_nbr = 0;

    printf ("I: 正在连接服务器 %s...\n", server [server_nbr]);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, server [server_nbr]);

    int sequence = 0;
    while (!zctx_interrupted) {
        //  发送请求并等待应答
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  轮询套接字
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL
            if (rc == -1)
                break;          //  中断
```

```
                //  处理应答
                if (items [0].revents & ZMQ_POLLIN) {
                    //  审核应答编号
                    char *reply = zstr_recv (client);
                    if (atoi (reply) == sequence) {
                        printf ("I: 服务端应答正常 (%s)\n", reply);
                        expect_reply = 0;
                        sleep (1);  //  每秒发送一个请求
                    }
                    else {
                        printf ("E: 错误的应答内容: %s\n",
                            reply);
                    }
                    free (reply);
                }
                else {
                    printf ("W: 服务器无响应，正在重试\n");
                    //  重开套接字
                    zsocket_destroy (ctx, client);
                    server_nbr = (server_nbr + 1) % 2;
                    zclock_sleep (SETTLE_DELAY);
                    printf ("I: 正在连接服务端 %s...\n",
                            server [server_nbr]);
                    client = zsocket_new (ctx, ZMQ_REQ);
                    zsocket_connect (client, server [server_nbr]);

                    //  使用新套接字重发请求
                    zstr_send (client, request);
                }
            }
        }
        zctx_destroy (&ctx);
        return 0;
}
```

运行以下命令进行测试，顺序随意：

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

可以将主机进程杀掉，测试故障恢复机制；再开启主机，杀掉备机，查看还原机制。要注意是由客户端触发这两个事件的。

下图展现了服务进程的状态图。绿色状态下会接收客户端请求，粉色状态会拒绝请求。事件指的是同伴的状态，所以“同伴激活态”指的是同伴机器告知我们它处于激活态。“客户请求”表示我们从客户端获得了请求，“客户投票”则指我们从客户端获得了请求并且同伴已经超时死亡。

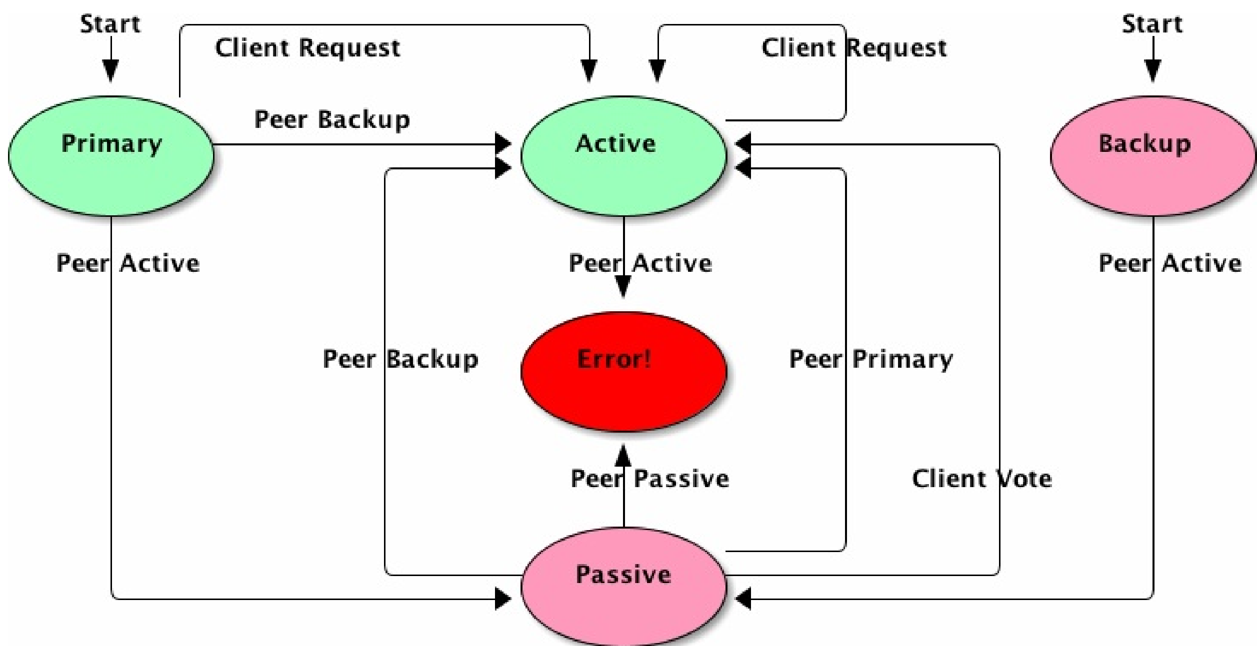


Figure 8 — Binary Star finite state machine

需要注意的是，服务进程使用PUB-SUB套接字进行状态交换，其它类型的套接字在这里不适用。比如，PUSH和DEALER套接字在没有节点相连的时候会发生阻塞；PAIR套接字不会在节点断开后进行重连；ROUTER套接字需要地址才能发送消息。

These are the main limitations of the Binary Star pattern:

- A server process cannot be part of more than one Binary Star pair.
- A primary server can have a single backup server, no more.
- The backup server cannot do useful work while in slave mode.
- The backup server must be capable of handling full application loads.
- Failover configuration cannot be modified at runtime.
- Client applications must do some work to benefit from failover.

双子星反应堆

我们可以将双子星模式打包成一个类似反应堆的类，供以后复用。在C语言中，我们使用czmq的zloop类，其他语言应该会有相应的实现。以下是C语言版的bstar接口：

```
// 创建双子星模式实例，使用本地（绑定）和远程（连接）端点来设置节点对。
bstar_t *bstar_new (int primary, char *local, char *remote);

// 销毁实例
void bstar_destroy (bstar_t **self_p);

// 返回底层的zloop反应堆，用以添加定时器、读取器、注册和取消等功能。
zloop_t *bstar_zloop (bstar_t *self);

// 注册投票读取器
int bstar_voter (bstar_t *self, char *endpoint, int type,
zloop_fn handler, void *arg);

// 注册状态机处理器
void bstar_new_master (bstar_t *self, zloop_fn handler, void *arg);
void bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg);

// 开启反应堆，当回调函数返回-1，或进程收到SIGINT、SIGTERM信号时中止。
int bstar_start (bstar_t *self);
```

以下是类的实现：

bstar: Binary Star core class in C

```
/*  =====================================================================
    bstar - Binary Star reactor

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "bstar.h"
```

```
//  服务器状态枚举
typedef enum {
    STATE_PRIMARY = 1,          //  主机，等待同伴连接
    STATE_BACKUP = 2,           //  备机，等待同伴连接
    STATE_ACTIVE = 3,           //  激活态，处理应用程序请求
    STATE_PASSIVE = 4           //  被动态，不接收请求
} state_t;

//  对话节点事件
typedef enum {
    PEER_PRIMARY = 1,           //  主机
    PEER_BACKUP = 2,            //  备机
    PEER_ACTIVE = 3,            //  激活态
    PEER_PASSIVE = 4,           //  被动态
    CLIENT_REQUEST = 5          //  客户端请求
} event_t;


//  发送状态信息的间隔时间
//  如果对方在两次心跳过后都没有应答，则视为断开
#define BSTAR_HEARTBEAT     1000        //  In msecs

//  类结构

struct _bstar_t {
    zctx_t *ctx;                //  私有上下文
    zloop_t *loop;              //  反应堆循环
    void *statepub;             //  状态发布者
    void *statesub;             //  状态订阅者
    state_t state;              //  当前状态
    event_t event;              //  当前事件
    int64_t peer_expiry;        //  判定节点死亡的时限
    zloop_fn *voter_fn;         //  投票套接字处理器
    void *voter_arg;            //  投票处理程序的参数
    zloop_fn *master_fn;        //  成为master时回调
    void *master_arg;           //  参数
    zloop_fn *slave_fn;         //  成为slave时回调
    void *slave_arg;            //  参数
};


//  ---------------------------------------------------------------------
//  执行有限状态机（将事件绑定至状态）；
//  发生异常时返回-1，正确时返回0。

static int
s_execute_fsm (bstar_t *self)
{
    int rc = 0;
    //  主机等待同伴连接
    //  该状态下接收CLIENT_REQUEST事件
    if (self->state == STATE_PRIMARY) {
```

```
        if (self->event == PEER_BACKUP) {
            zclock_log ("I: 已连接至备机（slave），可以作为master运行。"
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self->master_a
        }
        else
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: 已连接至备机（master），可以作为slave运行。"
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg
        }
        else
        if (self->event == CLIENT_REQUEST) {
            zclock_log ("I: 收到客户端请求，可作为master运行。");
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self->master_a
        }
    }
    else
    //  备机等待同伴连接
    //  该状态下拒绝CLIENT_REQUEST事件
    if (self->state == STATE_BACKUP) {
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: 已连接至主机（master），可以作为slave运行。"
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg
        }
        else
        if (self->event == CLIENT_REQUEST)
            rc = -1;
    }
    else
    //  服务器处于激活态
    //  该状态下接受CLIENT_REQUEST事件
    //  只有服务器死亡才会离开激活态
    if (self->state == STATE_ACTIVE) {
        if (self->event == PEER_ACTIVE) {
            //  若出现两台master，则抛出异常
            zclock_log ("E: 严重错误：双master。正在退出。");
            rc = -1;
        }
    }
    else
    //  服务器处于被动态
    //  若同伴已死，CLIENT_REQUEST事件将触发故障恢复
    if (self->state == STATE_PASSIVE) {
        if (self->event == PEER_PRIMARY) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            zclock_log ("I: 主机（slave）正在重启，可作为master运行。");
```

```
                self->state = STATE_ACTIVE;
            }
            else
            if (self->event == PEER_BACKUP) {
                //  同伴正在重启 - 转为激活态，同伴将转为被动态。
                zclock_log ("I: 备机（slave）正在重启，可作为master运行。");
                self->state = STATE_ACTIVE;
            }
            else
            if (self->event == PEER_PASSIVE) {
                //  若出现两台slave，集群将无响应
                zclock_log ("E: 严重错误：双slave。正在退出");
                rc = -1;
            }
            else
            if (self->event == CLIENT_REQUEST) {
                //  若心跳超时，同伴将成为master；
                //  此行为由客户端请求触发。
                assert (self->peer_expiry > 0);
                if (zclock_time () >= self->peer_expiry) {
                    //  同伴已死，转为激活态。
                    zclock_log ("I: 故障恢复，可作为master运行。");
                    self->state = STATE_ACTIVE;
                }
                else
                    //  同伴还在，拒绝请求。
                    rc = -1;
            }
            //  触发状态更改事件处理函数
            if (self->state == STATE_ACTIVE && self->master_fn)
                (self->master_fn) (self->loop, NULL, self->master_arg);
        }
        return rc;
}


//  ---------------------------------------------------------------------
//  反应堆事件处理程序

//  发送状态信息
int s_send_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    zstr_sendf (self->statepub, "%d", self->state);
    return 0;
}

//  接收状态信息，启动有限状态机
int s_recv_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    char *state = zstr_recv (socket);
    if (state) {
```

```
        self->event = atoi (state);
        self->peer_expiry = zclock_time () + 2 * BSTAR_HEARTBEAT;
        free (state);
    }
    return s_execute_fsm (self);
}

//  收到应用程序请求，判断是否接收
int s_voter_ready (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    //  如果能够处理请求，则调用函数
    self->event = CLIENT_REQUEST;
    if (s_execute_fsm (self) == 0) {
        puts ("CLIENT REQUEST");
        (self->voter_fn) (self->loop, socket, self->voter_arg);
    }
    else {
        //  销毁等待中的消息
        zmsg_t *msg = zmsg_recv (socket);
        zmsg_destroy (&msg);
    }
    return 0;
}


//  ---------------------------------------------------------------
//  构造函数

bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    //  初始化双子星
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    //  创建状态PUB套接字
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);

    //  创建状态SUB套接字
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->statesub, remote);

    //  设置基本的反应堆事件处理器
    zloop_timer (self->loop, BSTAR_HEARTBEAT, 0, s_send_state, self
    zloop_reader (self->loop, self->statesub, s_recv_state, self);
```

```
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
bstar_destroy (bstar_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        bstar_t *self = *self_p;
        zloop_destroy (&self->loop);
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  返回底层zloop对象，用以添加额外的定时器、阅读器等。

zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}


//  ---------------------------------------------------------------------
//  创建套接字，连接至本地端点，注册成为阅读器；
//  只有当有限状态机允许时才会读取该套接字；
//  从该套接字获得的消息将作为一次“投票”；
//  我们要求双子星模式中只有一个“投票”套接字。

int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn hand
             void *arg)
{
    //  保存原始的回调函数和参数，稍后使用
    void *socket = zsocket_new (self->ctx, type);
    zsocket_bind (socket, endpoint);
    assert (!self->voter_fn);
    self->voter_fn = handler;
    self->voter_arg = arg;
    return zloop_reader (self->loop, socket, s_voter_ready, self);
}

//  ---------------------------------------------------------------------
//  注册状态变化事件处理器
```

```
void
bstar_new_master (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->master_fn);
    self->master_fn = handler;
    self->master_arg = arg;
}

void
bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->slave_fn);
    self->slave_fn = handler;
    self->slave_arg = arg;
}


//  ---------------------------------------------------------------------
//  启用或禁止跟踪信息
void bstar_set_verbose (bstar_t *self, Bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}


//  ---------------------------------------------------------------------
//  开启反应堆，当回调函数返回-1，或进程收到SIGINT、SIGTERM信号时中止。

int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    return zloop_start (self->loop);
}
```

这样一来，我们的服务端代码会变得非常简短：

bstarsrv2: Binary Star server, using core class in C

```
//
//  双子星模式服务端，使用bstar反应堆
//

//  直接编译，不建类库
#include "bstar.c"

//  Echo service
int s_echo (zloop_t *loop, void *socket, void *arg)
{
    zmsg_t *msg = zmsg_recv (socket);
    zmsg_send (&msg, socket);
    return 0;
}

int main (int argc, char *argv [])
{
    //  命令行参数可以为：
    //      -p  作为主机启动, at tcp://localhost:5001
    //      -b  作为备机启动, at tcp://localhost:5002
    bstar_t *bstar;
    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: 主机master，等待备机（slave）连接。\n");
        bstar = bstar_new (BSTAR_PRIMARY,
            "tcp://*:5003", "tcp://localhost:5004");
        bstar_voter (bstar, "tcp://*:5001", ZMQ_ROUTER, s_echo, NUl
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: 备机slave，等待主机（master）连接。\n");
        bstar = bstar_new (BSTAR_BACKUP,
            "tcp://*:5004", "tcp://localhost:5003");
        bstar_voter (bstar, "tcp://*:5002", ZMQ_ROUTER, s_echo, NUl
    }
    else {
        printf ("Usage: bstarsrvs { -p | -b }\n");
        exit (0);
    }
    bstar_start (bstar);
    bstar_destroy (&bstar);
    return 0;
}
```

无中间件的可靠性（自由者模式）

我们讲了那么多关于中间件的示例，好像有些违背"ZMQ是无中间件"的说法。但要知道在现实生活中，中间件一直是让人又爱又恨的东西。实践中的很多消息架构能都在使用中间件进行分布式架构的搭建，所以说最终的决定还是需要你自己去权衡

的。这也是为什么虽然我能驾车10分钟到一个大型商场里购买五箱音量，但我还是会选择走10分钟到楼下的便利店里去买。这种出于经济方面的考虑（时间、精力、成本等）不仅在日常生活中很常见，在软件架构中也很重要。

这就是为什么ZMQ不会强制使用带有中间件的架构，但仍提供了像内置装置这样的中间件供编程人员自由选用。

这一节我们会打破以往使用中间件进行可靠性设计的架构，转而使用点对点架构，即自由者模式，来进行可靠的消息传输。我们的示例程序会是一个名称解析服务。ZMQ中的一个常见问题是：我们如何得知需要连接的端点？在代码中直接写入TCP/IP地址肯定是不合适的；使用配置文件会造成管理上的不便。试想一下，你要在上百台计算机中进行配置，只是为了让它们知道google.com的IP地址是74.125.230.82。

一个ZMQ的名称解析服务需要实现的功能有：

- 将逻辑名称解析为一个或多个端点地址，包括绑定端和连接端。实际使用时，名称服务会提供一组端点。
- 允许我们在不同的环境下，即开发环境和生产环境，进行解析；
- 该服务必须是可靠的，否则应用程序将无法连接到网络。

为管家模式提供名称解析服务会很有用，虽然将代理程序的端点对外暴露也很简单，但是如果用好名称解析服务，那它将成为唯一一个对外暴露的接口，将更便于管理。

我们需要处理的故障类型有：服务崩溃或重启、服务过载、网络因素等。为获取可靠性，我们必须建立一个服务群，当某个服务端崩溃后，客户端可以连接其他的服务端。实践中，两个服务端就已经足够了，但事实上服务端的数量可以是任意个。

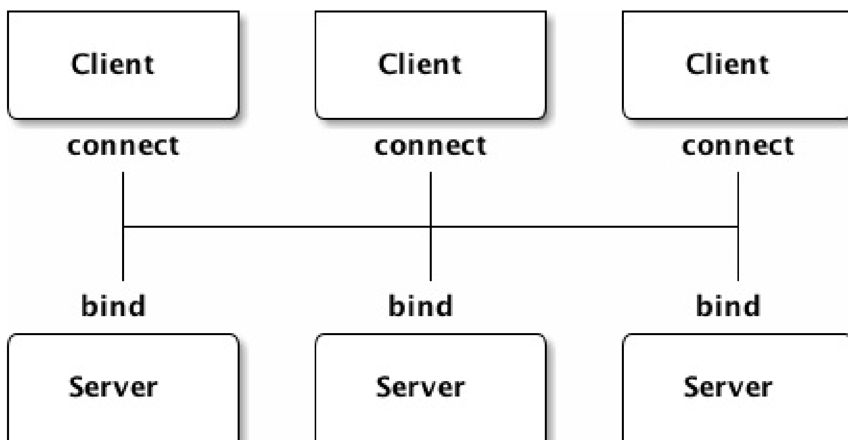


Figure 9 — The Freelance Pattern

在这个架构中，大量客户端和少量服务端进行通信，服务端将套接字绑定至单独的端口，这和管家模式中的代理有很大不同。对于客户端来说，它有这样几种选择：

- 客户端可以使用REQ套接字和懒惰海盗模式，但需要有一个机制防止客户端不断地请求已停止的服务端。

- 客户端可以使用DEALER套接字，向所有的服务端发送请求。很简单，但并不太妙；

- 客户端使用ROUTER套接字，连接特定的服务端。但客户端如何得知服务端的套接字标识呢？一种方式是让服务端主动连接客户端（很复杂），或者将服务端标识写入代码进行固化（很混乱）。

模型一：简单重试

让我们先尝试简单的方案，重写懒惰海盗模式，让其能够和多个服务端进行通信。启动服务端时用命令行参数指定端口。然后启动多个服务端。

flserver1: Freelance server, Model One in C

```
//
//  自由者模式 - 服务端 - 模型1
//  提供echo服务
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: echo服务端点： %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *msg = zmsg_recv (server);
        if (!msg)
            break;          //  中断
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: 中断\n");

    zctx_destroy (&ctx);
    return 0;
}
```

启动客户端，指定一个或多个端点：

flclient1: Freelance client, Model One in C

```
//
//  自由者模式 - 客户端 - 模型1
//  使用REQ套接字请求一个或多个服务端
//
#include "czmq.h"
```

```
#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       //  尝试次数


static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
{
    printf ("I: 在端点 %s 上尝试请求echo服务...\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  发送请求，并等待应答
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  关闭套接字
    zsocket_destroy (ctx, client);
    return reply;
}


int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
    else
    if (endpoints == 1) {
        //  若只有一个端点，则尝试N次
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  成功
            printf ("W: 没有收到 %s 的应答, 准备重试...\n", endpoint);
        }
    }
    else {
        //  若有多个端点，则每个尝试一次
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_n
```

```
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: 没有收到 %s 的应答\n", endpoint);
        }
    }
    if (reply)
        printf ("服务运作正常\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

可用如下命令运行：

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

客户端的核心机制是懒惰海盗模式，即获得一次成功的应答后就结束。会有两种情况：

- 如果只有一个服务端，客户端会再尝试N次后停止，这和懒惰海盗模式的逻辑一致；
- 如果有多个服务端，客户端会每个尝试一次，收到应答后停止。

这种机制补充了海盗模式，使其能够克服只有一个服务端的情况。

但是，这种设计无法在现实程序中使用：当有很多客户端连接了服务端，而主服务端崩溃了，那所有客户端都需要在超时后才能继续执行。

模型二：批量发送

下面让我们使用DEALER套接字。我们的目标是能再最短的时间里收到一个应答，不能受主服务端崩溃的影响。可以采取以下措施：

- 连接所有的服务端；
- 当有请求时，一次性发送给所有的服务端；
- 等待第一个应答；
- 忽略其他应答。

这样设计客户端时，当发送请求后，所有的服务端都会收到这个请求，并返回应答。如果某个服务端断开连接了，ZMQ可能会将请求发给其他服务端，导致某些服务端会收到两次请求。

更麻烦的是客户端无法得知应答的数量，容易发生混乱。

我们可以为请求进行编号，忽略不匹配的应答。我们要对服务端进行改造，返回的消息中需要包含请求编号： **flserver2: Freelance server, Model Two in C**

```
//
//  自由者模式 - 服务端 - 模型2
//  返回带有请求编号的OK信息
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: 服务已就绪 %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *request = zmsg_recv (server);
        if (!request)
            break;          //  中断
        //  判断请求内容是否正确
        assert (zmsg_size (request) == 2);

        zframe_t *address = zmsg_pop (request);
        zmsg_destroy (&request);

        zmsg_t *reply = zmsg_new ();
        zmsg_add (reply, address);
        zmsg_addstr (reply, "OK");
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

客户端代码：

flclient2: Freelance client, Model Two in C

```
//
//  自由者模式 - 客户端 - 模型2
//  使用DEALER套接字发送批量消息
```

```
//
#include "czmq.h"

//  超时时间
#define GLOBAL_TIMEOUT 2500

//  将客户端API封装成一个类

#ifdef __cplusplus
extern "C" {
#endif

//  声明类结构
typedef struct _flclient_t flclient_t;

flclient_t *
    flclient_new (void);
void
    flclient_destroy (flclient_t **self_p);
void
    flclient_connect (flclient_t *self, char *endpoint);
zmsg_t *
    flclient_request (flclient_t *self, zmsg_t **request_p);

#ifdef __cplusplus
}
#endif


int main (int argc, char *argv [])
{
    if (argc == 1) {
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    //  创建自由者模式客户端
    flclient_t *client = flclient_new ();

    //  连接至各个端点
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    //  发送一组请求，并记录时间
    int requests = 10000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flclient_request (client, &request);
        if (!reply) {
            printf ("E: 名称解析服务不可用，正在退出\n");
            break;
```

```
        }
        zmsg_destroy (&reply);
    }
    printf ("平均请求时间: %d 微秒\n",
        (int) (zclock_time () - start) / 10);

    flclient_destroy (&client);
    return 0;
}



//  --------------------------------------------------------------------
//  类结构

struct _flclient_t {
    zctx_t *ctx;        //  上下文
    void *socket;       //  用于和服务端通信的DEALER套接字
    size_t servers;     //  以连接的服务端数量
    uint sequence;      //  已发送的请求数
};


//  --------------------------------------------------------------------
//  Constructor

flclient_t *
flclient_new (void)
{
    flclient_t
        *self;

    self = (flclient_t *) zmalloc (sizeof (flclient_t));
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_DEALER);
    return self;
}

//  --------------------------------------------------------------------
//  析构函数

void
flclient_destroy (flclient_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flclient_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  -------------------------------------------------------------------
//  连接至新的服务端端点

void
flclient_connect (flclient_t *self, char *endpoint)
{
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}

//  -------------------------------------------------------------------
//  发送请求，接收应答
//  发送后销毁请求

zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    //  向消息添加编号和空帧
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);
    zmsg_pushstr (request, "");

    //  向所有已连接的服务端发送请求
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->socket);
    }
    //  接收来自任何服务端的应答
    //  因为我们可能poll多次，所以每次都进行计算
    zmsg_t *reply = NULL;
    uint64_t endtime = zclock_time () + GLOBAL_TIMEOUT;
    while (zclock_time () < endtime) {
        zmq_pollitem_t items [] = { { self->socket, 0, ZMQ_POLLIN,
        zmq_poll (items, 1, (endtime - zclock_time ()) * ZMQ_POLL_M
        if (items [0].revents & ZMQ_POLLIN) {
            //  应答内容是 [empty][sequence][OK]
            reply = zmsg_recv (self->socket);
            assert (zmsg_size (reply) == 3);
            free (zmsg_popstr (reply));
            char *sequence = zmsg_popstr (reply);
            int sequence_nbr = atoi (sequence);
            free (sequence);
            if (sequence_nbr == self->sequence)
                break;
        }
    }
```

```
    zmsg_destroy (request_p);
    return reply;
}
```

几点说明：

- 客户端被封装成了一个API类，将复杂的代码都包装了起来。
- 客户端会在几秒之后放弃寻找可用的服务端；
- 客户端需要创建一个合法的REP信封，所以需要添加一个空帧。

程序中，客户端发出了1万次名称解析请求（虽然是假的），并计算平均耗费时间。在我的测试机上，有一个服务端时，耗时60微妙；三个时80微妙。

该模型的优缺点是：

- 优点：简单，容易理解和编写；
- 优点：它工作迅速，有重试机制；
- 缺点：占用了额外的网络带宽；
- 缺点：我们不能为服务端设置优先级，如主服务、次服务等；
- 缺点：服务端不能同时处理多个请求。

Model Three - Complex and Nasty

批量发送模型看起来不太真实，那就让我们来探索最后这个极度复杂的模型。很有可能在编写完之后我们又会转而使用批量发送，哈哈，这就是我的作风。

我们可以将客户端使用的套接字更换为ROUTER，让我们能够向特定的服务端发送请求，停止向已死亡的服务端发送请求，从而做得尽可能地智能。我们还可以将服务端的套接字更换为ROUTER，从而突破单线程的瓶颈。

但是，使用ROUTER-ROUTER套接字连接两个瞬时套接字是不可行的，节点只有在收到第一条消息时才会为对方生成套接字标识。唯一的方法是让其中一个节点使用持久化的套接字，比较好的方式是让客户端知道服务端的标识，即服务端作为持久化的套接字。

为了避免产生新的配置项，我们直接使用服务端的端点作为套接字标识。

回想一下ZMQ套接字标识是如何工作的。服务端的ROUTER套接字为自己设置一个标识（在绑定之前），当客户端连接时，通过一个握手的过程来交换双方的标识。客户端的ROUTER套接字会先发送一条空消息，服务端为客户端生成一个随机的UUID。然后，服务端会向客户端发送自己的标识。

这样一来，客户端就可以将消息发送给特定的服务端了。不过还有一个问题：我们不知道服务端会在什么时候完成这个握手的过程。如果服务端是在线的，那可能几毫秒就能完成。如果不在线，那可能需要很久很久。

这里有一个矛盾：我们需要知道服务端何时连接成功且能够开始工作。自由者模式不像中间件模式，它的服务端必须要先发送请求后才能的应答。所以在服务端发送消息给客户端之前，客户端必须要先请求服务端，这看似是不可能的。

我有一个解决方法，那就是批量发送。这里发送的不是真正的请求，而是一个试探性的心跳（PING-PONG）。当收到应答时，就说明对方是在线的。

下面让我们制定一个协议，来定义自由者模式是如何传递这种心跳的：

- http://rfc.zeromq.org/spec:10

实现这个协议的服务端很方便，下面就是经过改造的echo服务：

flserver3: Freelance server, Model Three in C

```
//
//  自由者模式 - 服务端 - 模型3
//  使用ROUTER-ROUTER套接字进行通信；单线程。
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    zctx_t *ctx = zctx_new ();

    //  准备服务端套接字，其标识和端点名相同
    char *bind_endpoint = "tcp://*:5555";
    char *connect_endpoint = "tcp://localhost:5555";
    void *server = zsocket_new (ctx, ZMQ_ROUTER);
    zmq_setsockopt (server,
        ZMQ_IDENTITY, connect_endpoint, strlen (connect_endpoint));
    zsocket_bind (server, bind_endpoint);
    printf ("I: 服务端已准备就绪 %s\n", bind_endpoint);

    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (server);
        if (verbose && request)
            zmsg_dump (request);
        if (!request)
            break;          //  中断

        //  Frame 0: 客户端标识
        //  Frame 1: 心跳，或客户端控制信息帧
        //  Frame 2: 请求内容
        zframe_t *address = zmsg_pop (request);
        zframe_t *control = zmsg_pop (request);
        zmsg_t *reply = zmsg_new ();
        if (zframe_streq (control, "PONG"))
            zmsg_addstr (reply, "PONG");
        else {
            zmsg_add (reply, control);
            zmsg_addstr (reply, "OK");
        }
        zmsg_destroy (&request);
        zmsg_push (reply, address);
```

```
            if (verbose && reply)
                zmsg_dump (reply);
            zmsg_send (&reply, server);
        }
        if (zctx_interrupted)
            printf ("W: 中断\n");

        zctx_destroy (&ctx);
        return 0;
    }
```

但是，自由者模式的客户端会变得大一写。为了清晰期间，我们将其拆分为两个类来实现。首先是在上层使用的程序：

flclient3: Freelance client, Model Three in C

```
//
//  自由者模式 - 客户端 - 模型3
//  使用flcliapi类来封装自由者模式
//
//  直接编译，不建类库
#include "flcliapi.c"

int main (void)
{
    //  创建自由者模式实例
    flcliapi_t *client = flcliapi_new ();

    //  链接至服务器端点
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");

    //  发送随机请求，计算时间
    int requests = 1000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flcliapi_request (client, &request);
        if (!reply) {
            printf ("E: 名称解析服务不可用，正在退出\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("平均执行时间： %d usec\n",
        (int) (zclock_time () - start) / 10);

    flcliapi_destroy (&client);
    return 0;
}
```

下面是该模式复杂的实现过程：

flcliapi: Freelance client API in C

```
/*  ================================================================
    flcliapi - Freelance Pattern agent class
    Model 3: uses ROUTER socket to address specific services

    ----------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org
```

```
    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    ===============================================================
*/

#include "flcliapi.h"

//  请求超时时间
#define GLOBAL_TIMEOUT  3000    //  msecs
//  心跳间隔
#define PING_INTERVAL   2000    //  msecs
//  判定服务死亡的时间
#define SERVER_TTL      6000    //  msecs


//  ===============================================================
//  同步部分，在应用程序层面运行

//  ---------------------------------------------------------------
//  类结构

struct _flcliapi_t {
    zctx_t *ctx;        //  上下文
    void *pipe;         //  用于和主线程通信的套接字
};

//  这是运行后台代理程序的线程
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);


//  ---------------------------------------------------------------
//  构造函数

flcliapi_t *
flcliapi_new (void)
{
    flcliapi_t
        *self;

    self = (flcliapi_t *) zmalloc (sizeof (flcliapi_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, flcliapi_agent, NULL);
```

```
    return self;
}

//  ---------------------------------------------------------------------
//  析构函数

void
flcliapi_destroy (flcliapi_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flcliapi_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
//  连接至新服务器端点
//  消息内容：[CONNECT][endpoint]

void
flcliapi_connect (flcliapi_t *self, char *endpoint)
{
    assert (self);
    assert (endpoint);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, endpoint);
    zmsg_send (&msg, self->pipe);
    zclock_sleep (100);      //  等待连接
}

//  ---------------------------------------------------------------------
//  发送并销毁请求，接收应答

zmsg_t *
flcliapi_request (flcliapi_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);

    zmsg_pushstr (*request_p, "REQUEST");
    zmsg_send (request_p, self->pipe);
    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *status = zmsg_popstr (reply);
        if (streq (status, "FAILED"))
            zmsg_destroy (&reply);
        free (status);
    }
    return reply;
```

```
}


//  =====================================================================
//  异步部分，在后台运行

//  ---------------------------------------------------------------------
//  单个服务端信息

typedef struct {
    char *endpoint;             //  服务端端点/套接字标识
    uint alive;                 //  是否在线
    int64_t ping_at;            //  下一次心跳时间
    int64_t expires;            //  过期时间
} server_t;

server_t *
server_new (char *endpoint)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));
    self->endpoint = strdup (endpoint);
    self->alive = 0;
    self->ping_at = zclock_time () + PING_INTERVAL;
    self->expires = zclock_time () + SERVER_TTL;
    return self;
}

void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->endpoint);
        free (self);
        *self_p = NULL;
    }
}

int
server_ping (char *key, void *server, void *socket)
{
    server_t *self = (server_t *) server;
    if (zclock_time () >= self->ping_at) {
        zmsg_t *ping = zmsg_new ();
        zmsg_addstr (ping, self->endpoint);
        zmsg_addstr (ping, "PING");
        zmsg_send (&ping, socket);
        self->ping_at = zclock_time () + PING_INTERVAL;
    }
    return 0;
}
```

```
int
server_tickless (char *key, void *server, void *arg)
{
    server_t *self = (server_t *) server;
    uint64_t *tickless = (uint64_t *) arg;
    if (*tickless > self->ping_at)
        *tickless = self->ping_at;
    return 0;
}


//  ---------------------------------------------------------------------
//  后台处理程序信息

typedef struct {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  用于应用程序通信的套接字
    void *router;               //  用于服务端通信的套接字
    zhash_t *servers;           //  已连接的服务端
    zlist_t *actives;           //  在线的服务端
    uint sequence;              //  请求编号
    zmsg_t *request;            //  当前请求
    zmsg_t *reply;              //  当前应答
    int64_t expires;            //  请求过期时间
} agent_t;

agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->router = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->servers = zhash_new ();
    self->actives = zlist_new ();
    return self;
}

void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->servers);
        zlist_destroy (&self->actives);
        zmsg_destroy (&self->request);
        zmsg_destroy (&self->reply);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  当服务端从列表中移除时，回调该函数。

static void
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

void
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);

    if (streq (command, "CONNECT")) {
        char *endpoint = zmsg_popstr (msg);
        printf ("I: connecting to %s...\n", endpoint);
        int rc = zmq_connect (self->router, endpoint);
        assert (rc == 0);
        server_t *server = server_new (endpoint);
        zhash_insert (self->servers, endpoint, server);
        zhash_freefn (self->servers, endpoint, s_server_free);
        zlist_append (self->actives, server);
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;
        free (endpoint);
    }
    else
    if (streq (command, "REQUEST")) {
        assert (!self->request);    //  遵循请求-应答循环
        //  将请求编号和空帧加入消息顶部
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        //  获取请求消息的所有权
        self->request = msg;
        msg = NULL;
        //  设置请求过期时间
        self->expires = zclock_time () + GLOBAL_TIMEOUT;
    }
    free (command);
    zmsg_destroy (&msg);
}

void
agent_router_message (agent_t *self)
{
    zmsg_t *reply = zmsg_recv (self->router);

    //  第一帧是应答的服务端标识
    char *endpoint = zmsg_popstr (reply);
    server_t *server =
```

```
        (server_t *) zhash_lookup (self->servers, endpoint);
    assert (server);
    free (endpoint);
    if (!server->alive) {
        zlist_append (self->actives, server);
        server->alive = 1;
    }
    server->ping_at = zclock_time () + PING_INTERVAL;
    server->expires = zclock_time () + SERVER_TTL;

    //  第二帧是应答的编号
    char *sequence = zmsg_popstr (reply);
    if (atoi (sequence) == self->sequence) {
        zmsg_pushstr (reply, "OK");
        zmsg_send (&reply, self->pipe);
        zmsg_destroy (&self->request);
    }
    else
        zmsg_destroy (&reply);
}


//  ---------------------------------------------------------------------
//  异步的后台代理会维护一个服务端池，处理请求和应答。

static void
flcliapi_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    zmq_pollitem_t items [] = {
        { self->pipe, 0, ZMQ_POLLIN, 0 },
        { self->router, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        //  计算超时时间
        uint64_t tickless = zclock_time () + 1000 * 3600;
        if (self->request
        &&  tickless > self->expires)
            tickless = self->expires;
        zhash_foreach (self->servers, server_tickless, &tickless);

        int rc = zmq_poll (items, 2,
            (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文对象被关闭

        if (items [0].revents & ZMQ_POLLIN)
            agent_control_message (self);

        if (items [1].revents & ZMQ_POLLIN)
            agent_router_message (self);
```

```
            //  如果我们需要处理一项请求，将其发送给下一个可用的服务端
            if (self->request) {
                if (zclock_time () >= self->expires) {
                    //  请求超时
                    zstr_send (self->pipe, "FAILED");
                    zmsg_destroy (&self->request);
                }
                else {
                    //  寻找可用的服务端
                    while (zlist_size (self->actives)) {
                        server_t *server =
                            (server_t *) zlist_first (self->actives);
                        if (zclock_time () >= server->expires) {
                            zlist_pop (self->actives);
                            server->alive = 0;
                        }
                        else {
                            zmsg_t *request = zmsg_dup (self->request);
                            zmsg_pushstr (request, server->endpoint);
                            zmsg_send (&request, self->router);
                            break;
                        }
                    }
                }
            }
            //  断开并删除已过期的服务端
            //  发送心跳给空闲服务器
            zhash_foreach (self->servers, server_ping, self->router);
        }
        agent_destroy (&self);
    }
```

这组API使用了较为复杂的机制，我们之前也有用到过：

异步后台代理

客户端API由两部分组成：同步的flcliapi类，运行于应用程序线程；异步的agent类，运行于后台线程。flcliapi和agent类通过一个inproc套接字互相通信。所有和ZMQ相关的内容都封装在API中。agent类实质上是作为一个迷你的代理程序在运行，负责在后台与服务端进行通信，只要我们发送请求，它就会设法连接一个服务器来处理请求。

连接等待机制

ROUTER套接字的特点之一是会直接丢弃无法路由的消息，这就意味着当与服务器建立了ROUTER-ROUTER连接后，如果立刻发送一条消息，该消息是会丢失的。flcliapi类则延迟了一会儿后再发送消息。之后的通信中，由于服务端套接字是持久的，客户端就不再丢弃消息了。

Ping silence

0MQ will queue messages for a dead server indefinitely. So if a client repeatedly PINGs a dead server, when that server comes back to life it'll get a whole bunch of PING messages all at once. Rather than continuing to ping a server we know is offline, we count on 0MQ's handling of durable sockets to deliver the old PING messages when the server comes back online. As soon as a server reconnects, it'll get PINGs from all clients that were connected to it, it'll PONG back, and those clients will recognize it as alive again.

调整轮询时间

在之前的示例程序中，我们一般会为轮询设置固定的超时时间（如1秒），这种做法虽然简单，但是对于用电较为敏感的设备来说（如笔记本电脑或手机）唤醒CPU是需要额外的电力的。所以，为了完美也好，好玩也好，我们这里调整了轮询时间，将其设置为到达过期时间时才超时，这样就能节省一部分轮询次数了。我们可以将过期时间放入一个列表中存储，方便查询。

总结

这一章中我们看到了很多可靠的请求-应答机制，每种机制都有其优劣性。大部分示例代码是可以直接用于生产环境的，不过还可以进一步优化。有两个模式会比较典型：使用了中间件的管家模式，以及未使用中间件的自由者模式。

第五章 高级发布-订阅模式

第三章和第四章讲述了ZMQ中请求-应答模式的一些高级用法。如果你已经能够彻底理解了，那我要说声恭喜。这一章我们会关注发布-订阅模式，使用上层模式封装，提升ZMQ发布-订阅模式的性能、可靠性、状态同步及安全机制。

本章涉及的内容有：

- 处理慢订阅者（自杀的蜗牛模式）
- 高速订阅者（黑箱模式）
- 构建一个共享键值缓存（克隆模式）

检测慢订阅者（自杀的蜗牛模式）

在使用发布-订阅模式的时候，最常见的问题之一是如何处理响应较慢的订阅者。理想状况下，发布者能以全速发送消息给订阅者，但现实中，订阅者会需要对消息做较长时间的处理，或者写得不够好，无法跟上发布者的脚步。

如何处理慢订阅者？最好的方法当然是让订阅者高效起来，不过这需要额外的工作。以下是一些处理慢订阅者的方法：

- 在发布者中贮存消息。这是Gmail的做法，如果过去的几小时里没有阅读邮件的话，它会把邮件保存起来。但在高吞吐量的应用中，发布者堆积消息往往会导致内存溢出，最终崩溃。特别是当同是有多个订阅者时，或者无法用磁盘来做一个缓冲，情况就会变得更为复杂。

- 在订阅者中贮存消息。这种做法要好的多，其实ZMQ默认的行为就是这样的。如果非得有一个人会因为内存溢出而崩溃，那也只会是订阅者，而非发布者，这挺公平的。然而，这种做法只对瞬间消息量很大的应用才合理，订阅者只是一时处理不过来，但最终会赶上进度。但是，这还是没有解决订阅者速度过慢的问题。

- 暂停发送消息。这也是Gmail的做法，当我的邮箱容量超过7.554GB时，新的邮件就会被Gmail拒收或丢弃。这种做法对发布者来说很有益，ZMQ中若设置了阈值（HWM），其默认行为也就是这样的。但是，我们仍不能解决慢订阅者的问题，我们只是让消息变得断断续续而已。

- 断开与满订阅者的连接。这是hotmail的做法，如果连续两周没有登录，它就会断开，这也是为什么我正在使用第十五个hotmail邮箱。不过这种方案在ZMQ里是行不通的，因为对于发布者而言，订阅者是不可见的，无法做相应处理。

看来没有一种经典的方式可以满足我们的需求，所以我们就要进行创新了。我们可以让订阅者自杀，而不仅仅是断开连接。这就是“自杀的蜗牛”模式。当订阅者发现自身运行得过慢时（对于慢速的定义应该是一个配置项，当达到这个标准时就大声地喊出来吧，让程序员知道），它会哀嚎一声，然后自杀。

订阅者如何检测自身速度过慢呢？一种方式是为消息进行编号，并在发布者端设置阈值。当订阅者发现消息编号不连续时，它就知道事情不对劲了。这里的阈值就是订阅者自杀的值。

这种方案有两个问题：一、如果我们连接的多个发布者，我们要如何为消息进行编号呢？解决方法是为每一个发布者设定一个唯一的编号，作为消息编号的一部分。二、如果订阅者使用ZMQ_SUBSRIBE选项对消息进行了过滤，那么我们精心设计的消息编号机制就毫无用处了。

有些情形不会进行消息的过滤，所以消息编号还是行得通的。不过更为普遍的解决方案是，发布者为消息标注时间戳，当订阅者收到消息时会检测这个时间戳，如果其差别达到某一个值，就发出警报并自杀。

当订阅者有自身的客户端或服务协议，需要保证最大延迟时间时，自杀的蜗牛模式会很合适。撤销一个订阅者也许并不是最周全的方案，但至少不会引发后续的问题。如果订阅者收到了过时的消息，那可能会对数据造成进一步的破坏，而且很难被发现。

以下是自杀的蜗牛模式的最简实现：

suisnail: Suicidal Snail in C

```
//
//  自杀的蜗牛模式
//
#include "czmq.h"

//  ---------------------------------------------------------------------
//  该订阅者会连接至发布者，接收所有的消息，
//  运行过程中它会暂停一会儿，模拟复杂的运算过程，
//  当发现收到的消息超过1秒的延迟时，就自杀。

#define MAX_ALLOWED_DELAY   1000    //  毫秒

static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  订阅所有消息
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  获取并处理消息
    while (1) {
        char *string = zstr_recv (subscriber);
        int64_t clock;
        int terms = sscanf (string, "%" PRId64, &clock);
        assert (terms == 1);
        free (string);

        //  自杀逻辑
        if (zclock_time () - clock > MAX_ALLOWED_DELAY) {
            fprintf (stderr, "E: 订阅者无法跟进, 取消中\n");
```

```
                break;
            }
            //  工作一定时间
            zclock_sleep (1 + randof (2));
        }
        zstr_send (pipe, "订阅者中止");
}


//  ---------------------------------------------------------------------
//  发布者每毫秒发送一条用时间戳标记的消息

static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  准备发布者
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (1) {
        //  发送当前时间（毫秒）给订阅者
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);            //  等待1毫秒
    }
}


//  下面的代码会启动一个订阅者和一个发布者，当订阅者死亡时停止运行
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 示例程序中的消息包含了系统当前的时间戳（毫秒）。在现实应用中，你应该

使用时间戳作为消息头，并提供消息内容。
- 示例程序中的发布者和订阅者是同一个进程的两个线程。在现实应用中，他们应该是两个不同的进程。示例中这么做只是为了演示的方便

高速订阅者（黑箱模式）

发布-订阅模式的一个典型应用场景是大规模分布式数据处理。如要处理从证券市场上收集到的数据，可以在证券交易系统上设置一个发布者，获取价格信息，并发送给一组订阅者。如果我们有很多订阅者，我们可以使用TCP。如果订阅者到达一定的量，那我们就应该使用可靠的广播协议，如pgm。

假设我们的发布者每秒产生10万条100个字节的消息。在剔除了不需要的市场信息后，这个比率还是比较合理的。现在我们需要记录一天的数据（8小时约有250GB），再将其传入一个模拟网络，即一组订阅者。虽然10万条数据对ZMQ来说很容易处理，但我们需要更高的速度。

假设我们有多台机器，一台做发布者，其他的做订阅者。这些机器都是8核的，发布者那台有12核。

在我们开始发布消息时，有两点需要注意：

1. 即便只是处理很少的数据，订阅者仍有可能跟不上发布者的速度；
2. 当处理到6M/s的数据量时，发布者和订阅者都有可能达到极限。

首先，我们需要将订阅者设计为一种多线程的处理程序，这样我们就能在一个线程中读取消息，使用其他线程来处理消息。一般来说，我们对每种消息的处理方式都是不同的。这样一来，订阅者可以对收到的消息进行一次过滤，如根据头信息来判别。当消息满足某些条件，订阅者会将消息交给worker处理。用ZMQ的语言来说，订阅者会将消息转发给worker来处理。

这样一来，订阅者看上去就像是一个队列装置，我们可以用各种方式去连接队列装置和worker。如我们建立单向的通信，每个worker都是相同的，可以使用PUSH和PULL套接字，分发的工作就交给ZMQ吧。这是最简单也是最快速的方式：

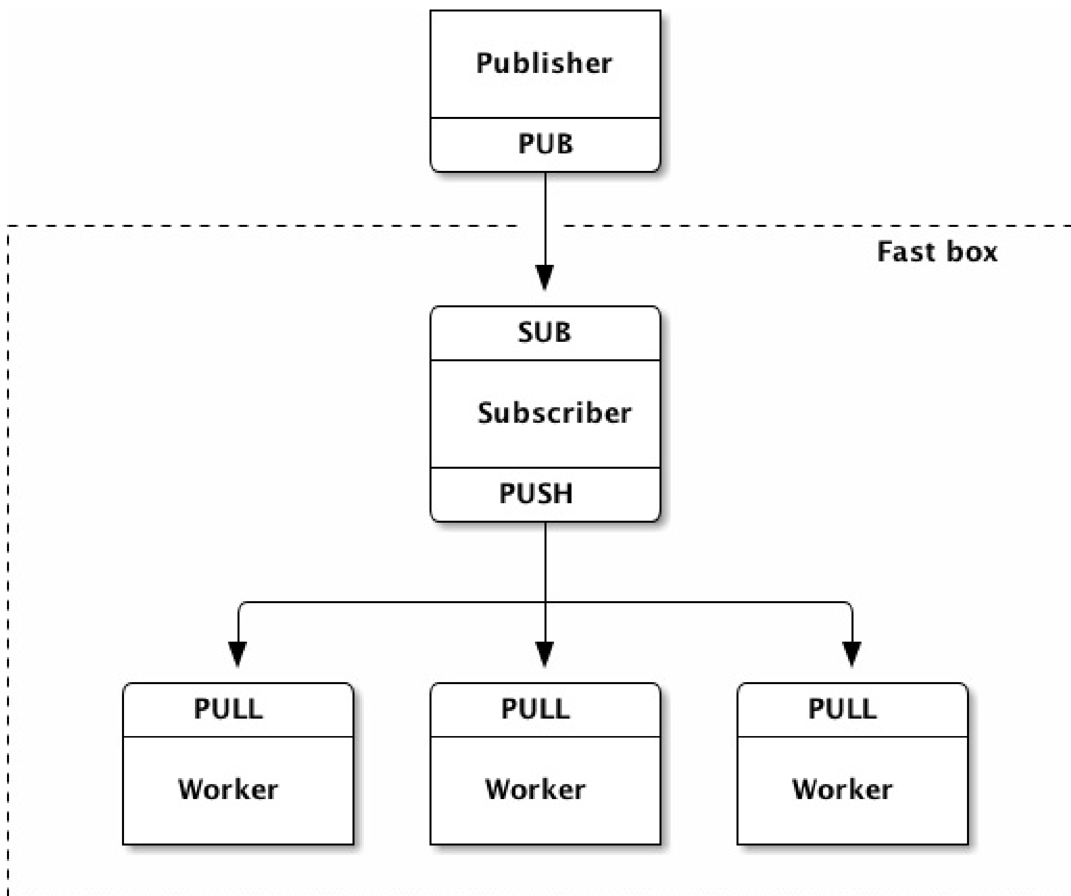


Figure 1 — Simple Black Box Pattern

订阅者和发布者之间的通信使用TCP或PGM协议，订阅者和worker的通信由于是在同一个进程中完成的，所以使用inproc协议。

下面我们看看如何突破瓶颈。由于订阅者是单线程的，当它的CPU占用率达到100%时，它无法使用其他的核心。单线程程序总是会遇到瓶颈的，不管是2M、6M还是更多。我们需要将工作量分配到不同的线程中去，并发地执行。

很多高性能产品使用的方案是分片，就是将工作量拆分成独立并行的流。如，一半的专题数据由一个流媒体传输，另一半由另一个流媒体传输。我们可以建立更多的流媒体，但如果CPU核心数不变，那就没有必要了。 让我们看看如何将工作量分片为两个流：

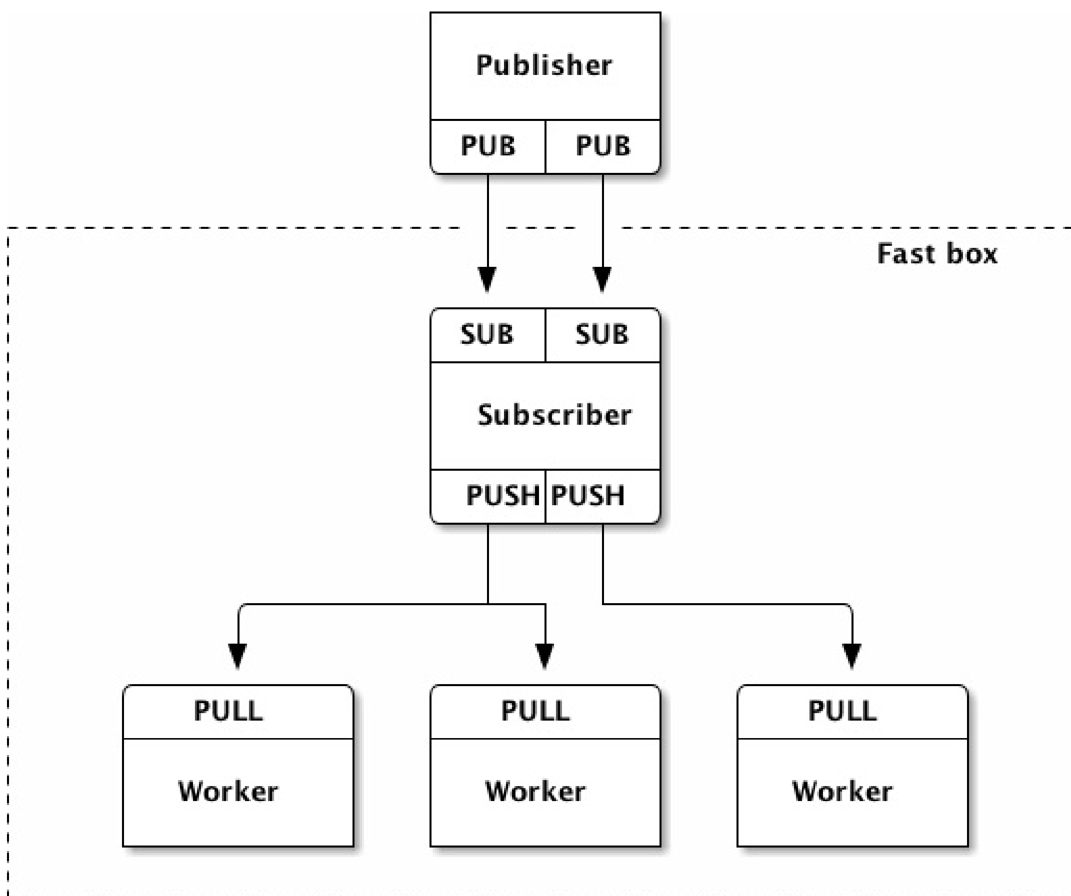


Figure 2 — Mad Black Box Pattern

要让两个流全速工作，需要这样配置ZMQ：

- 使用两个I/O线程，而不是一个；
- 使用两个独立的网络接口；
- 每个I/O线程绑定至一个网络接口；
- 两个订阅者线程，分别绑定至一个核心；
- 使用两个SUB套接字；
- 剩余的核心供worker使用；
- worker线程同时绑定至两个订阅者线程的PUSH套接字。

创建的线程数量应和CPU核心数一致，如果我们建立的线程数量超过核心数，那其处理速度只会减少。另外，开放多个I/O线程也是没有必要的。

共享键值缓存（克隆模式）

发布-订阅模式和无线电广播有些类似，在你收听之前发送的消息你将无从得知，收到消息的多少又会取决于你的接收能力。让人吃惊的是，对于那些追求完美的工程师来说，这种机器恰恰符合他们的需求，且广为传播，成为现实生活中分发消息的最佳机制。想想非死不可、推特、BBS新闻、体育新闻等应用就知道了。

但是，在很多情形下，可靠的发布-订阅模式同样是有价值的。正如我们讨论请求-应答模式一样，我们会根据“故障”来定义“可靠性”，下面几项便是发布-订阅模式中可能发生的故障：

- 订阅者连接太慢，因此没有收到发布者最初发送的消息；
- 订阅者速度太慢，同样会丢失消息；
- 订阅者可能会断开，其间的消息也会丢失。

还有一些情况我们碰到的比较少，但不是没有：

- 订阅者崩溃、重启，从而丢失了所有已收到的消息；
- 订阅者处理消息的速度过慢，导致消息在队列中堆砌并溢出；
- 因网络过载而丢失消息（特别是PGM协议下的连接）；
- 网速过慢，消息在发布者处溢出，从而崩溃。

其实还会有其他出错的情况，只是以上这些在现实应用中是比较典型的。

我们已经有方法解决上面的某些问题了，比如对于慢速订阅者可以使用自杀的蜗牛模式。但是，对于其他的问题，我们最后能有一个可复用的框架来编写可靠的发布-订阅模式。

难点在于，我们并不知道目标应用程序会怎样处理这些数据。它们会进行过滤、只处理一部分消息吗？它们是否会将消息记录起来供日后使用？它们是否会将消息转发给其下的worker进行处理？需要考虑的情况实在太多了，每种情况都有其所谓的可靠性。

所以，我们将问题抽象出来，供多种应用程序使用。这种抽象应用我们称之为共享的键值缓存，它的功能是通过唯一的键名存储二进制数据块。

不要将这个抽象应用和分布式哈希表混淆起来，它是用来解决节点在分布式网络中相连接的问题的；也不要和分布式键值表混淆，它更像是一个NoSQL数据库。我们要建立的应用是将内存中的状态可靠地传递给一组客户端，它要做到的是：

- 客户端可以随时加入网络，并获得服务端当前的状态；
- 任何客户端都可以改变键值缓存（插入、更新、删除）；
- 将这种变化以最短的延迟可靠地传达给所有的客户端；
- 能够处理大量的客户端，成百上千。

克隆模式的要点在于客户端会反过来和服务端进行通信，这在简单的发布-订阅模式中并不常见。所以我这里使用“服务端”、“客户端”而不是“发布者”、“订阅者”这两个词。我们会使用发布-订阅模式作为核心消息模式，不过还需要夹杂其他模式。

分发键值更新事件

我们会分阶段实施克隆模式。首先，我们看看如何从服务器发送键值更新事件给所有的客户端。我们将第一章中使用的天气服务模型进行改造，以键值对的方式发送信息，并让客户端使用哈希表来保存：

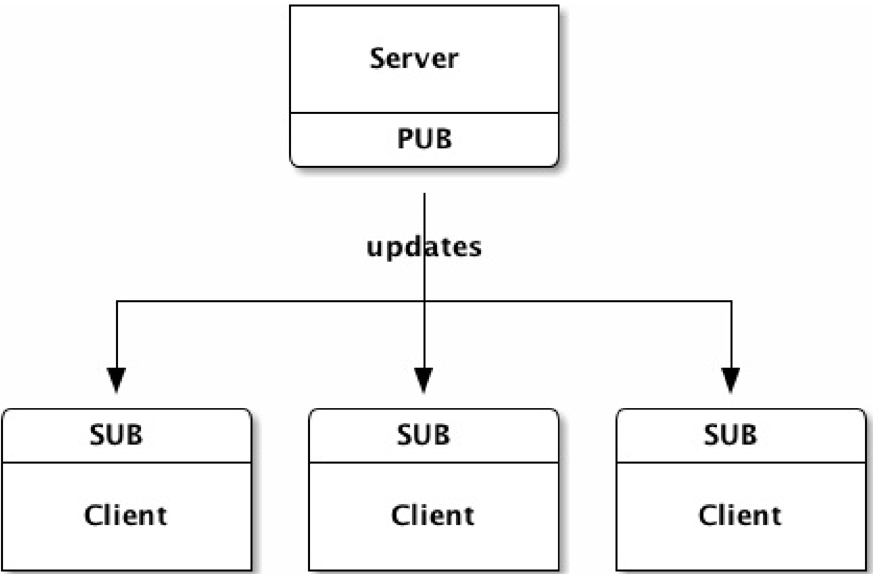


Figure 3 — Simplest Clone Model

以下是服务端代码：

clonesrv1: Clone server, Model One in C

```
//
//  克隆模式服务端模型1
//

//  让我们直接编译，不生成类库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和PUB套接字
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");
    zclock_sleep (200);

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    while (!zctx_interrupted) {
        //  使用键值对分发消息
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_store   (&kvmsg, kvmap);
    }
    printf (" 已中止\n已发送 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

以下是客户端代码：

clonecli1: Clone client, Model One in C

```
//
//  克隆模式客户端模型1
//

//  让我们直接编译，不生成类库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *updates = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (updates, "tcp://localhost:5556");

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;

    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (updates);
        if (!kvmsg)
            break;          //  中断
        kvmsg_store (&kvmsg, kvmap);
        sequence++;
    }
    printf (" 已中断\n收到 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 所有复杂的工作都在kvmsg类中完成了，这个类能够处理键值对类型的消息对象，其实质上是一个ZMQ多帧消息，共有三帧：键（ZMQ字符串）、编号（64位，按字节顺序排列）、二进制体（保存所有附加信息）。
- 服务端随机生成消息，使用四位数作为键，这样可以模拟大量而不是过量的哈希表（1万个条目）。
- 服务端绑定套接字后会等待200毫秒，以避免订阅者连接延迟而丢失数据的问题。我们会在后面的模型中解决这一点。
- 我们使用“发布者”和“订阅者”来命名程序中使用的套接字，这样可以避免和后续模型中的其他套接字发生混淆。

以下是kvmsg的代码，已经经过了精简： **kvsimple: Key-value message class in C**

```
/*  =====================================================================
    kvsimple - simple key-value message class for example applicati

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
```

```
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    ================================================================
*/

#include "kvsimple.h"
#include "zlist.h"

//  键是一个短字符串
#define KVMSG_KEY_MAX   255

//  消息被格式化成三帧
//  frame 0: 键（ZMQ字符串）
//  frame 1: 编号（8个字节，按顺序排列）
//  frame 2: 内容（二进制数据块）
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_BODY      2
#define KVMSG_FRAMES    3

//  类结构
struct _kvmsg {
    //  消息中某帧是否存在
    int present [KVMSG_FRAMES];
    //  对应的ZMQ消息帧
    zmq_msg_t frame [KVMSG_FRAMES];
    //  将键转换为C语言字符串
    char key [KVMSG_KEY_MAX + 1];
};


//  ---------------------------------------------------------------
//  构造函数，设置编号

kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
```

```
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    kvmsg_set_sequence (self, sequence);
    return self;
}


//  ---------------------------------------------------------------
//  析构函数

//  释放消息中的帧，可供zhash_freefn()函数调用
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  销毁消息中的帧
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  释放对象本身
        free (self);
    }
}

void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------
//  从套接字中读取键值消息，返回kvmsg实例

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  读取所有帧，出错则销毁对象
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
```

```
            zmq_msg_init (&self->frame [frame_nbr]);
            self->present [frame_nbr] = 1;
            if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1
                kvmsg_destroy (&self);
                break;
            }
            //  验证多帧消息
            int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
            if (zsockopt_rcvmore (socket) != rcvmore) {
                kvmsg_destroy (&self);
                break;
            }
        }
        return self;
}


//  ---------------------------------------------------------------
//  向套接字发送键值对消息，不检验消息帧的内容

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}


//  ---------------------------------------------------------------
//  从消息中获取键值，不存在则返回NULL

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
```

```
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息的编号

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  返回消息内容，不存在则返回NULL

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息内容的大小

size_t
kvmsg_size (kvmsg_t *self)
```

```
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  设置消息的键

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}


//  ---------------------------------------------------------------------
//  设置消息的编号

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}


//  ---------------------------------------------------------------------
//  设置消息内容
```

```
void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}


//  ---------------------------------------------------------------------
//  使用printf()格式设置消息键

void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}


//  ---------------------------------------------------------------------
//  使用springf()格式设置消息内容

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}


//  ---------------------------------------------------------------------
//  若kvmsg结构的键值均存在，则存入哈希表；
//  如果kvmsg结构已没有引用，则自动销毁和释放。

void
```

```
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (self->present [FRAME_KEY]
        &&  self->present [FRAME_BODY]) {
            zhash_update (hash, kvmsg_key (self), self);
            zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
        }
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  将消息内容打印至标准错误输出，用以调试和跟踪

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self)
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}


//  ---------------------------------------------------------------------
//  测试用例

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");
```

```
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  测试简单消息的发送和接受
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  关闭并销毁所有对象
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

我们会在下文编写一个更为完整的kvmsg类，可以用到现实环境中。

客户端和服务端都会维护一个哈希表，但这个模型需要所有的客户端都比服务端启动得早，而且不能崩溃，这显然不能满足可靠性的要求。

创建快照

为了让后续连接的（或从故障中恢复的）客户端能够获取服务器上的状态信息，需要让它在连接时获取一份快照。正如我们将“消息”的概念简化为“已编号的键值对”，我们也可以将“状态”简化为“一个哈希表”。为获取服务端状态，客户端会打开一个REQ套接字进行请求：

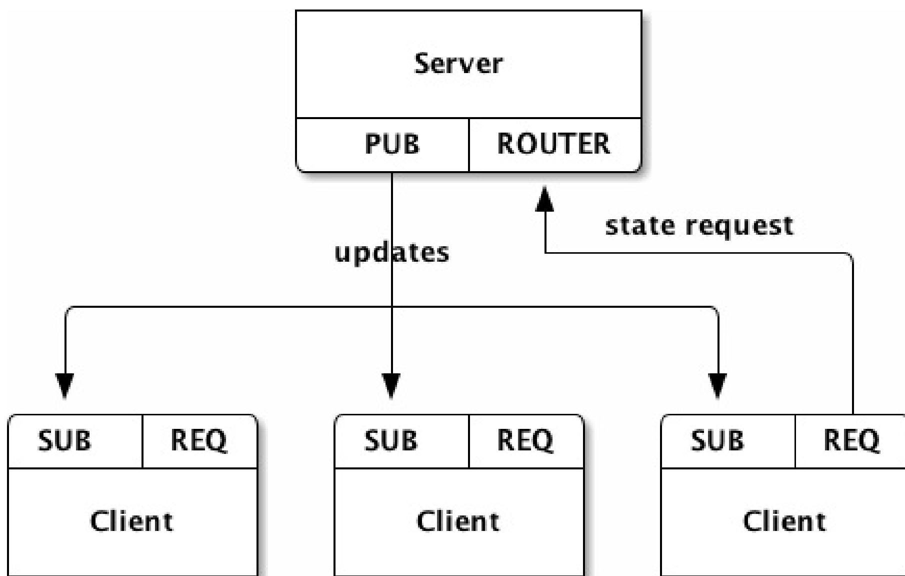


Figure 4 — State Replication

我们需要考虑时间的问题，因为生成快照是需要一定时间的，我们需要知道应从哪个更新事件开始更新快照，服务端是不知道何时有更新事件的。一种方法是先开始订阅消息，收到第一个消息之后向服务端请求“将该条更新之前的所有内容发送给”。这样一来，服务器需要为每一次更新保存一份快照，这显然是不现实的。

所以，我们会在客户端用以下方式进行同步：

- 客户端开始订阅服务器的更新事件，然后请求一份快照。这样就能保证这份快照是在上一次更新事件之后产生的。
- 客户端开始等待服务器的快照，并将更新事件保存在队列中，做法很简单，不要从套接字中读取消息就可以了，ZMQ会自动将这些消息保存起来，这时不应设置阈值（HWM）。
- 当客户端获取到快照后，它将再次开始读取更新事件，但是需要丢弃那些早于快照生成时间的事件。如快照生成时包含了200次更新，那客户端会从第201次更新开始读取。
- 随后，客户端就会用更新事件去更新自身的状态了。

这是一个比较简单的模型，因为它用到了ZMQ消息队列的机制。服务端代码如下：

clonesrv2: Clone server, Model Two in C

```
//
//  克隆模式 - 服务端 - 模型2
//

//  让我们直接编译，不创建类库
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);
static void state_manager (void *args, zctx_t *ctx, void *pipe);
```

```
int main (void)
{
    //  准备套接字和上下文
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");

    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    //  开启状态管理器，并等待同步信号
    void *updates = zthread_fork (ctx, state_manager, NULL);
    free (zstr_recv (updates));

    while (!zctx_interrupted) {
        //  分发键值消息
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_send     (kvmsg, updates);
        kvmsg_destroy (&kvmsg);
    }
    printf (" 已中断\n已发送 %d 条消息\n", (int) sequence);
    zctx_destroy (&ctx);
    return 0;
}

//  快照请求方信息
typedef struct {
    void *socket;           //  用于发送快照的ROUTER套接字
    zframe_t *identity;     //  请求方的标识
} kvroute_t;

//  发送快照中单个键值对
//  使用kvmsg对象作为载体
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  先发送接收方标识
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

//  该线程维护服务端状态，并处理快照请求。
//
static void
state_manager (void *args, zctx_t *ctx, void *pipe)
{
```

```
    zhash_t *kvmap = zhash_new ();

    zstr_send (pipe, "READY");
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");

    zmq_pollitem_t items [] = {
        { pipe, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    int64_t sequence = 0;       //  当前快照版本
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1 && errno == ETERM)
            break;              //  上下文异常

        //  等待主线程的更新事件
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (pipe);
            if (!kvmsg)
                break;          //  中断
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        //  执行快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求内容在第二帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照给客户端
            kvroute_t routing = { snapshot, identity };

            //  逐项发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识，内含快照版本号
            printf ("正在发送快照，版本号 %d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
```

```
    }
    zhash_destroy (&kvmap);
}
```

以下是客户端代码：

clonecli2: Clone client, Model Two in C

```
//
// 克隆模式 - 客户端 - 模型2
//

// 让我们直接编译，不生成类库
#include "kvsimple.c"

int main (void)
{
    // 准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");

    zhash_t *kvmap = zhash_new ();

    // 获取快照
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          // 中断
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("已获取快照，版本号=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          // 完成
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    // 应用队列中的更新事件，丢弃过时事件
    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);
        if (!kvmsg)
            break;          // 中断
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 客户端使用两个线程，一个用来生成随机的更新事件，另一个用来管理状态。两者之间使用PAIR套接字通信。可能你会考虑使用SUB套接字，但是“慢连接”的问题会影响到程序运行。PAIR套接字会让两个线程严格同步的。
- 我们在updates套接字上设置了阈值（HWM），避免更新服务内存溢出。在inproc协议的连接中，阈值是两端套接字阈值的加和，所以要分别设置。
- 客户端比较简单，用C语言编写，大约60行代码。大多数工作都在kvmsg类中完成了，不过总的来说，克隆模式实现起来还是比较简单的。
- 我们没有用特别的方式来序列化状态内容。键值对用kvmsg对象表示，保存在一个哈希表中。在不同的时间请求状态时会得到不同的快照。
- 我们假设客户端只和一个服务进行通信，而且服务必须是正常运行的。我们暂不考虑如何从服务崩溃的情形中恢复过来。

现在，这两段程序都还没有真正地工作起来，但已经能够正确地同步状态了。这是一个多种消息模式的混合体：进程内的PAIR、发布-订阅、ROUTER-DEALER等。

重发键值更新事件

第二个模型中，键值更新事件都来自于服务器，构成了一个中心化的模型。但是我们需要的是一个能够在客户端进行更新的缓存，并能同步到其他客户端中。这时，服务端只是一个无状态的中间件，带来的好处有：

- 我们不用太过关心服务端的可靠性，因为即使它崩溃了，我们仍能从客户端中获取完整的数据。
- 我们可以使用键值缓存在动态节点之间分享数据。

客户端的键值更新事件会通过PUSH-PULL套接字传达给服务端：

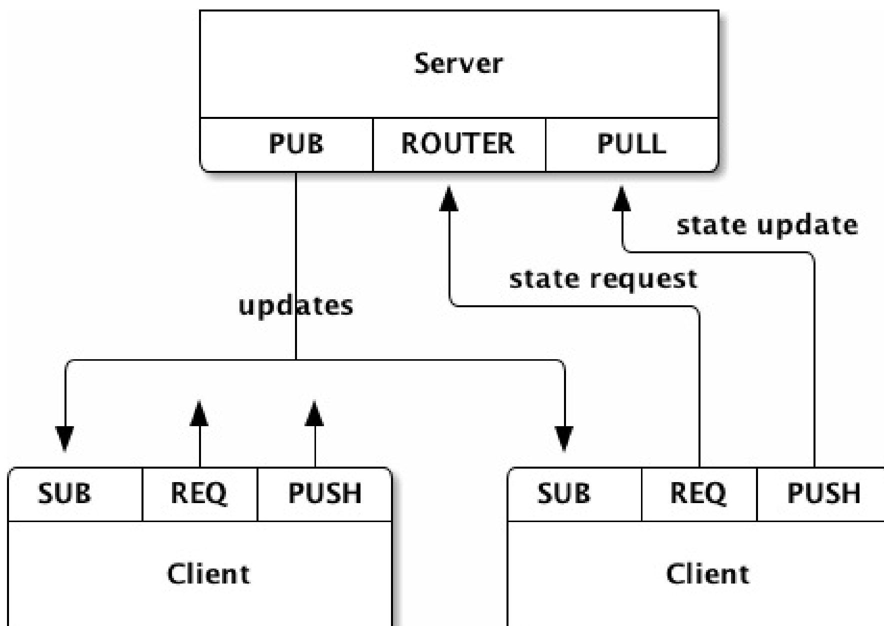


Figure 5 — Republishing Updates

我们为什么不让客户端直接将更新信息发送给其他客户端呢？虽然这样做可以减少延迟，但是就无法为更新事件添加自增的唯一编号了。很多应用程序都需要更新事件以某种方式排序，只有将消息发给服务端，由服务端分发更新消息，才能保证更新事件的顺序。

有了唯一的编号后，客户端还能检测到更多的故障：网络堵塞或队列溢出。如果客户端发现消息输入流有一段空白，它能采取措施。可能你会觉得此时让客户端通知服务端，让它重新发送丢失的信息，可以解决问题。但事实上没有必要这么做。消息流的空挡表示网络状况不好，如果再进行这样的请求，只会让事情变得更糟。所以一般的做法是由客户端发出警告，并停止运行，等到有专人来维护后再继续工作。 我们开始创建在客户端进行状态更新的模型。以下是客户端代码：

clonesrv3: Clone server, Model Three in C

```
//
//  克隆模式 服务端 模型3
//

//  直接编译，不创建类库
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  快照请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
} kvroute_t;


int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  执行来自客户端的更新事件
```

```
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  中断
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: 发布更新事件 %5d\n", (int) sequence);
        }
        //  响应快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求内容在消息的第二帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照
            kvroute_t routing = { snapshot, identity };

            //  逐条发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识和编号
            printf ("I: 正在发送快照，版本号：%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    printf (" 已中断\n已处理 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  发送一条键值对状态给套接字，使用kvmsg对象保存键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
```

```
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}
```

以下是客户端代码：

clonecli3: Clone client, Model Three in C

```
//
//  克隆模式 - 客户端 - 模型3
//

//  直接编译，不创建类库
#include "kvsimple.c"

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  获取状态快照
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  中断
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: 已收到快照，版本号：%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  完成
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
```

```
                tickless = 0;
            int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;              //  上下文被关闭

            if (items [0].revents & ZMQ_POLLIN) {
                kvmsg_t *kvmsg = kvmsg_recv (subscriber);
                if (!kvmsg)
                    break;          //  中断

                //  丢弃过时消息，包括心跳
                if (kvmsg_sequence (kvmsg) > sequence) {
                    sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, kvmap);
                    printf ("I: 收到更新事件：%d\n", (int) sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
            //  创建一个随机的更新事件
            if (zclock_time () >= alarm) {
                kvmsg_t *kvmsg = kvmsg_new (0);
                kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
                kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
                kvmsg_send     (kvmsg, publisher);
                kvmsg_destroy (&kvmsg);
                alarm = zclock_time () + 1000;
            }
        }
        printf (" 已准备\n收到 %d 条消息\n", (int) sequence);
        zhash_destroy (&kvmap);
        zctx_destroy (&ctx);
        return 0;
    }
```

几点说明：

- 服务端整合为一个线程，负责收集来自客户端的更新事件并转发给其他客户端。它使用PULL套接字获取更新事件，ROUTER套接字处理快照请求，以及PUB套接字发布更新事件。
- 客户端会每隔1秒左右发送随机的更新事件给服务端，现实中这一动作由应用程序触发。

子树克隆

现实中的键值缓存会越变越多，而客户端可能只会需要部分缓存。我们可以使用子树的方式来实现：客户端在发送快照请求时告诉服务端它需要的子树，在订阅更新事件时也指明子树。

关于子树的语法有很多，一种是“分层路径”结构，另一种是“主题树”：

- 分层路径：/some/list/of/paths
    - 主题树：some.list.of.topics

这里我们会使用分层路径结构，以此扩展服务端和客户端，进行子树操作。维护多个子树其实并不太困难，因此我们不在这里演示。

下面是服务端代码，由模型3衍化而来：

clonesrv4: Clone server, Model Four in C

```
//
//  克隆模式 服务端 模型4
//

//  直接编译，不创建类库
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  快照请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
    char *subtree;          //  指定的子树
} kvroute_t;


int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  执行来自客户端的更新事件
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
```

```
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: 发布更新事件 %5d\n", (int) sequence);
        }
        //  响应快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  请求内容在消息的第二帧中
            char *request = zstr_recv (snapshot);
            char *subtree = NULL;
            if (streq (request, "ICANHAZ?")) {
                free (request);
                subtree = zstr_recv (snapshot);
            }
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照
            kvroute_t routing = { snapshot, identity, subtree };

            //  逐条发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识和编号
            printf ("I: 正在发送快照，版本号：%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    printf (" 已中断\n已处理 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  发送一条键值对状态给套接字，使用kvmsg对象保存键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
```

```
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0)
        //  先发送接收方的标识
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

下面是客户端代码：

clonecli4: Clone client, Model Four in C

```
//
//  克隆模式 - 客户端 - 模型4
//

//  直接编译，不创建类库
#include "kvsimple.c"

#define SUBTREE "/client/"

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsockopt_set_subscribe (subscriber, SUBTREE);
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  获取状态快照
    int64_t sequence = 0;
    zstr_sendm (snapshot, "ICANHAZ?");
    zstr_send  (snapshot, SUBTREE);
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: 已收到快照，版本号：%d\n", (int) sequence);
```

```
                kvmsg_destroy (&kvmsg);
                break;          //  Done
            }
            kvmsg_store (&kvmsg, kvmap);
        }

        int64_t alarm = zclock_time () + 1000;
        while (!zctx_interrupted) {
            zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0
            int tickless = (int) ((alarm - zclock_time ()));
            if (tickless < 0)
                tickless = 0;
            int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;              //  上下文被关闭

            if (items [0].revents & ZMQ_POLLIN) {
                kvmsg_t *kvmsg = kvmsg_recv (subscriber);
                if (!kvmsg)
                    break;          //  中断

                //  丢弃过时消息，包括心跳
                if (kvmsg_sequence (kvmsg) > sequence) {
                    sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, kvmap);
                    printf ("I: 收到更新事件：%d\n", (int) sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
            //  创建一个随机的更新事件
            if (zclock_time () >= alarm) {
                kvmsg_t *kvmsg = kvmsg_new (0);
                kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000))
                kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
                kvmsg_send     (kvmsg, publisher);
                kvmsg_destroy (&kvmsg);
                alarm = zclock_time () + 1000;
            }
        }
        printf (" 已准备\n收到 %d 条消息\n", (int) sequence);
        zhash_destroy (&kvmap);
        zctx_destroy (&ctx);
        return 0;
    }
```

瞬间值

瞬间值指的是那些会立刻过期的值。如果你用克隆模式搭建一个类似DNS的服务时，就可以用瞬间值来模拟动态DNS解析了。当节点连接网络，对外发布它的地址，并不断地更新地址。如果节点断开连接，则它的地址也会失效。

瞬间值可以和会话（session）联系起来，当会话结束时，瞬间值也就失效了。克隆模式中，会话是由客户端定义的，并会在客户端断开连接时消亡。

更简单的方法是为每一个瞬间值设定一个过期时间，客户端会不断延长这个时间，当断开连接时这个时间将得不到更新，服务器就会自动将其删除。

我们会用这种简单的方法来实现瞬间值，因为太过复杂的方法可能不值当，它们的差别仅在性能上体现。如果客户端有很多瞬间值，那为每个值设定过期时间是恰当的；如果瞬间值到达一定的量，那最好还是将其和会话相关联，统一进行过期处理。

首先，我们需要设法在键值对消息中加入过期时间。我们可以增加一个消息帧，但这样一来每当我们需要增加消息内容时就需要修改kvmsg类库了，这并不合适。所以，我们一次性增加一个“属性”消息帧，用于添加不同的消息属性。

其次，我们需要设法删除这条数据。目前为止服务端和客户端会盲目地增改哈希表中的数据，我们可以这样定义：当消息的值是空的，则表示删除这个键的数据。

下面是一个更为完整的kvmsg类代码，它实现了“属性”帧，以及一个UUID帧，我们后面会用到。该类还会负责处理值为空的消息，达到删除的目的：

kvmsg: Key-value message class - full in C

```
/*  =====================================================================
    kvmsg - key-value message class for example applications

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/
```

```
#include "kvmsg.h"
#include <uuid/uuid.h>
#include "zlist.h"

//  键是短字符串
#define KVMSG_KEY_MAX   255

//  消息包含五帧
//  frame 0: 键(ZMQ字符串)
//  frame 1: 编号(8个字节，按顺序排列)
//  frame 2: UUID(二进制块，16个字节)
//  frame 3: 属性(ZMQ字符串)
//  frame 4: 值(二进制块)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_UUID      2
#define FRAME_PROPS     3
#define FRAME_BODY      4
#define KVMSG_FRAMES    5

//  类结构
struct _kvmsg {
    //  帧是否存在
    int present [KVMSG_FRAMES];
    //  对应消息帧
    zmq_msg_t frame [KVMSG_FRAMES];
    //  键，C语言字符串格式
    char key [KVMSG_KEY_MAX + 1];
    //  属性列表，key=value形式
    zlist_t *props;
    size_t props_size;
};


//  将属性列表序列化为字符串
static void
s_encode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    if (self->present [FRAME_PROPS])
        zmq_msg_close (msg);

    zmq_msg_init_size (msg, self->props_size);
    char *prop = zlist_first (self->props);
    char *dest = (char *) zmq_msg_data (msg);
    while (prop) {
        strcpy (dest, prop);
        dest += strlen (prop);
        *dest++ = '\n';
        prop = zlist_next (self->props);
    }
    self->present [FRAME_PROPS] = 1;
}
```

```
//  从字符串中解析属性列表
static void
s_decode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    self->props_size = 0;
    while (zlist_size (self->props))
        free (zlist_pop (self->props));

    size_t remainder = zmq_msg_size (msg);
    char *prop = (char *) zmq_msg_data (msg);
    char *eoln = memchr (prop, '\n', remainder);
    while (eoln) {
        *eoln = 0;
        zlist_append (self->props, strdup (prop));
        self->props_size += strlen (prop) + 1;
        remainder -= strlen (prop) + 1;
        prop = eoln + 1;
        eoln = memchr (prop, '\n', remainder);
    }
}


//  ---------------------------------------------------------------------
//  构造函数，指定消息编号

kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    self->props = zlist_new ();
    kvmsg_set_sequence (self, sequence);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

//  释放内存函数，供zhash_free_fn()调用
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  释放所有消息帧
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
```

```
                zmq_msg_close (&self->frame [frame_nbr]);

        //  释放属性列表
        while (zlist_size (self->props))
            free (zlist_pop (self->props));
        zlist_destroy (&self->props);

        //  释放对象本身
        free (self);
    }
}

void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  复制kvmsg对象

kvmsg_t *
kvmsg_dup (kvmsg_t *self)
{
    kvmsg_t *kvmsg = kvmsg_new (0);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr]) {
            zmq_msg_t *src = &self->frame [frame_nbr];
            zmq_msg_t *dst = &kvmsg->frame [frame_nbr];
            zmq_msg_init_size (dst, zmq_msg_size (src));
            memcpy (zmq_msg_data (dst),
                    zmq_msg_data (src), zmq_msg_size (src));
            kvmsg->present [frame_nbr] = 1;
        }
    }
    kvmsg->props = zlist_copy (self->props);
    return kvmsg;
}


//  ---------------------------------------------------------------------
//  从套接字总读取键值对，返回kvmsg实例

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
```

```
    kvmsg_t *self = kvmsg_new (0);

    //  读取所有帧，若有异常则直接返回空
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1
            kvmsg_destroy (&self);
            break;
        }
        //  验证多帧消息
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    if (self)
        s_decode_props (self);
    return self;
}


//  ---------------------------------------------------------------
//  向套接字发送键值对消息，空消息也发送

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    s_encode_props (self);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}


//  ---------------------------------------------------------------
//  返回消息的键

char *
```

```
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}


//  ---------------------------------------------------------------
//  返回消息的编号

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}


//  ---------------------------------------------------------------
//  返回消息的UUID

byte *
kvmsg_uuid (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_UUID]
    &&  zmq_msg_size (&self->frame [FRAME_UUID]) == sizeof (uuid_t)
```

```
        return (byte *) zmq_msg_data (&self->frame [FRAME_UUID]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息的内容

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息内容的长度

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  设置消息的键

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}


//  ---------------------------------------------------------------------
//  设置消息的编号
```

```
void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}


//  ---------------------------------------------------------------
//  生成并设置消息的UUID

void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
    self->present [FRAME_UUID] = 1;
}


//  ---------------------------------------------------------------
//  设置消息的内容

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
```

```
    memcpy (zmq_msg_data (msg), body, size);
}


//  ---------------------------------------------------------------------
//  使用printf()格式设置消息的键
void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}


//  ---------------------------------------------------------------------
//  使用printf()格式设置消息内容

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}


//  ---------------------------------------------------------------------
//  获取消息属性，无则返回空字符串

char *
kvmsg_get_prop (kvmsg_t *self, char *name)
{
    assert (strchr (name, '=') == NULL);
    char *prop = zlist_first (self->props);
    size_t namelen = strlen (name);
    while (prop) {
        if (strlen (prop) > namelen
        &&  memcmp (prop, name, namelen) == 0
        &&  prop [namelen] == '=')
            return prop + namelen + 1;
        prop = zlist_next (self->props);
    }
```

```
    return "";
}


//  ---------------------------------------------------------------------
//  设置消息属性
//  属性名称不能包含=号，值的最大长度是255

void
kvmsg_set_prop (kvmsg_t *self, char *name, char *format, ...)
{
    assert (strchr (name, '=') == NULL);

    char value [255 + 1];
    va_list args;
    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);

    //  分配空间
    char *prop = malloc (strlen (name) + strlen (value) + 2);

    //  删除已存在的属性
    sprintf (prop, "%s=", name);
    char *existing = zlist_first (self->props);
    while (existing) {
        if (memcmp (prop, existing, strlen (prop)) == 0) {
            self->props_size -= strlen (existing) + 1;
            zlist_remove (self->props, existing);
            free (existing);
            break;
        }
        existing = zlist_next (self->props);
    }
    //  添加新属性
    strcat (prop, value);
    zlist_append (self->props, prop);
    self->props_size += strlen (prop) + 1;
}


//  ---------------------------------------------------------------------
//  在哈希表中保存kvmsg对象
//  当kvmsg对象不再被使用时进行释放操作；
//  若传入的值为空，则删除该对象。

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
```

```
        assert (self);
        if (kvmsg_size (self)) {
            if (self->present [FRAME_KEY]
            &&  self->present [FRAME_BODY]) {
                zhash_update (hash, kvmsg_key (self), self);
                zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
            }
        }
        else
            zhash_delete (hash, kvmsg_key (self));

        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  将消息内容输出到标准错误输出

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self)
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        if (zlist_size (self->props)) {
            fprintf (stderr, "[");
            char *prop = zlist_first (self->props);
            while (prop) {
                fprintf (stderr, "%s;", prop);
                prop = zlist_next (self->props);
            }
            fprintf (stderr, "]");
        }
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}


//  ---------------------------------------------------------------------
//  测试用例
```

```
int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  测试简单消息的收发
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    // 测试带有属性的消息的收发
    kvmsg = kvmsg_new (2);
    kvmsg_set_prop (kvmsg, "prop1", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value2");
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_destroy (&kvmsg);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
```

```
    assert (streq (kvmsg_key (kvmsg), "key"));
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    kvmsg_destroy (&kvmsg);

    //  关闭并销毁所有对象
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

客户端模型5和模型4没有太大区别，只是kvmsg类库变了。在更新消息的时候还需要添加一个过期时间的属性：

```
kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

服务端模型5有较大的变化，我们会用反应堆来代替轮询，这样就能混合处理定时事件和套接字事件了，只是在C语言中是比较麻烦的。下面是代码：

clonesrv5: Clone server, Model Five in C

```
//
//  克隆模式 - 服务端 - 模型5
//

//  直接编译，不建类库
#include "kvmsg.c"

//  反应堆处理器
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);

//  服务器属性
typedef struct {
    zctx_t *ctx;                //  上下文
    zhash_t *kvmap;             //  键值对存储
    zloop_t *loop;              //  zloop反应堆
    int port;                   //  主端口
    int64_t sequence;           //  更新事件编号
    void *snapshot;             //  处理快照请求
    void *publisher;            //  发布更新事件
    void *collector;            //  从客户端收集接收更新事件
} clonesrv_t;


int main (void)
{
```

```
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));

    self->port = 5556;
    self->ctx = zctx_new ();
    self->kvmap = zhash_new ();
    self->loop = zloop_new ();
    zloop_set_verbose (self->loop, FALSE);

    //  打开克隆模式服务端套接字
    self->snapshot  = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_PULL);
    zsocket_bind (self->snapshot,  "tcp://*:%d", self->port);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  注册反应堆处理程序
    zloop_reader (self->loop, self->snapshot, s_snapshots, self);
    zloop_reader (self->loop, self->collector, s_collector, self);
    zloop_timer  (self->loop, 1000, 0, s_flush_ttl, self);

    //  运行反应堆，直至中断
    zloop_start (self->loop);

    zloop_destroy (&self->loop);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);
    return 0;
}


//  ---------------------------------------------------------------------
//  发送快照内容

static int s_send_single (char *key, void *data, void *args);

//  请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
    char *subtree;          //  子树信息
} kvroute_t;

static int
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  请求位于消息第二帧
        char *request = zstr_recv (snapshot);
```

```
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: 错误的请求，程序中止\n");

        if (subtree) {
            //  发送状态快照
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  发送结束符和版本号
            zclock_log ("I: 正在发送快照，版本号：%d", (int) self->seq
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}


//  每次发送一个快照键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0)
        //  先发送接收方标识
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}


//  ---------------------------------------------------------------
//  收集更新事件

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
```

```
    kvmsg_t *kvmsg = kvmsg_recv (collector);
    if (kvmsg) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
        if (ttl)
            kvmsg_set_prop (kvmsg, "ttl",
                "%" PRId64, zclock_time () + ttl * 1000);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布更新事件 %d", (int) self->sequence);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  删除过期的瞬间值

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  删除过期的键值对，并广播该事件
static int
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 发布删除事件 %d", (int) self->sequence);
    }
    return 0;
}
```

克隆服务器的可靠性

克隆模型1至5相对比较简单，下面我们会探讨一个非常复杂的模型。可以发现，为了构建可靠的消息队列，我们需要花费非常多的精力。所以我们经常会问：有必要这么做吗？如果说你能够接受可靠性不够高的、或者说已经足够好的架构，那恭喜你，你在成本和收益之间找到了平衡。虽然我们会偶尔丢失一些消息，但从经济的角度来说还是合理的。不管怎样，下面我们就来介绍这个复杂的模型。

在模型3中，你会关闭和重启服务，这会导致数据的丢失。任何后续加入的客户端只能得到重启之后的那些数据，而非所有的。下面就让我们想办法让克隆模式能够承担服务器重启的故障。

以下列举我们需要处理的问题：

- 克隆服务器进程崩溃并自动或手工重启。进程丢失了所有数据，所以必须从别处进行恢复。
- 克隆服务器硬件故障，长时间不能恢复。客户端需要切换至另一个可用的服务端。
- 克隆服务器从网络上断开，如交换机发生故障等。它会在某个时点重连，但期间的数据就需要替代的服务器负责处理。

第一步我们需要增加一个服务器。我们可以使用第四章中提到的双子星模式，它是一个反应堆，而我们的程序经过整理后也是一个反应堆，因此可以互相协作。

我们需要保证更新事件在主服务器崩溃时仍能保留，最简单的机制就是同时发送给两台服务器。

备机就可以当做一台客户端来运行，像其他客户端一样从主机获取更新事件。同时它又能从客户端获取更新事件——虽然不应该以此更新数据，但可以先暂存起来。

所以，相较于模型5，模型6中引入了以下特性：

- 客户端发送更新事件改用PUB-SUB套接字，而非PUSH-PULL。原因是PUSH套接字会在没有接收方时阻塞，且会进行负载均衡——我们需要两台服务器都接收到消息。我们会在服务器端绑定SUB套接字，在客户端连接PUB套接字。
- 我们在服务器发送给客户端的更新事件中加入心跳，这样客户端可以知道主机是否已死，然后切换至备机。
- 我们使用双子星模式的bstar反应堆类来创建主机和备机。双子星模式中需要有一个“投票”套接字，来协助判定对方节点是否已死。这里我们使用快照请求来作为“投票”。
- 我们将为所有的更新事件添加UUID属性，它由客户端生成，服务端会将其发布给所有客户端。
- 备机将维护一个“待处理列表”，保存来自客户端、尚未由服务端发布的更新事件；或者反过来，来自服务端、尚未从客户端收到的更新事件。这个列表从旧到新排列，这样就能方便地从顶部删除消息。

我们可以为客户端设计一个有限状态机，它有三种状态：

- 客户端打开并连接了套接字，然后向服务端发送快照请求。为了避免消息风暴，它只会请求两次。
- 客户端等待快照应答，如果获得了则保存它；如果没有获得，则向第二个服务器发送请求。
- 客户端收到快照，便开始等待更新事件。如果在一定时间内没有收到服务端响应，则会连接第二个服务端。

客户端会一直循环下去，可能在程序刚启动时，部分客户端会试图连接主机，部分连接备机，相信双子星模式会很好地处理这一情况的。

我们可以将客户端状态图绘制出来：

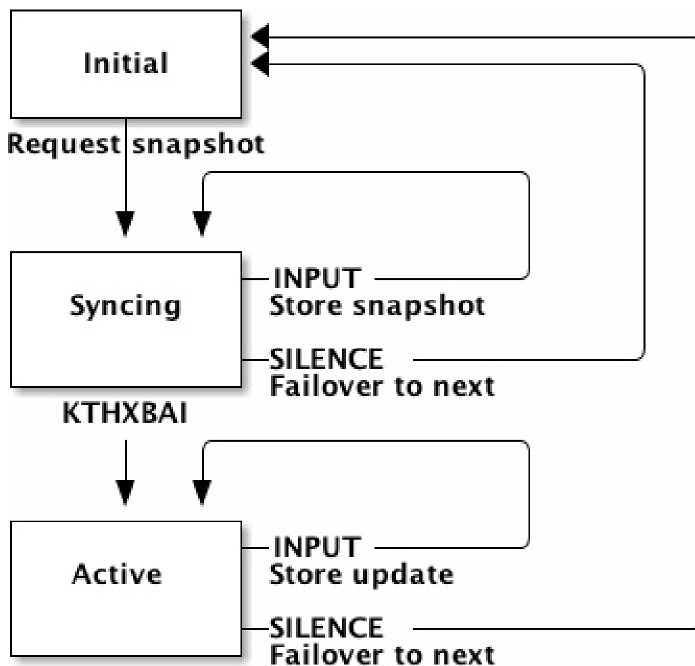


Figure 6 — Clone client FSM

故障恢复的步骤如下：

- 客户端检测到主机不再发送心跳，因此转而连接备机，并请求一份新的快照；
- 备机开始接收快照请求，并检测到主机死亡，于是开始作为主机运行；
- 备机将待处理列表中的更新事件写入自身状态中，然后开始处理快照请求。

当主机恢复连接时：

- 启动为slave状态，并作为克隆模式客户端连接备机；
- 同时，使用SUB套接字从客户端接收更新事件。

我们做两点假设：

- 至少有一台主机会继续运行。如果两台主机都崩溃了，那我们将丢失所有的服务端数据，无法恢复。
- 不同的客户端不会同时更新同一个键值对。客户端的更新事件会先后到达两个服务器，因此更新的顺序可能会不一致。单个客户端的更新事件到达两台服务器的顺序是相同的，所以不用担心。

下面是整体架构图：

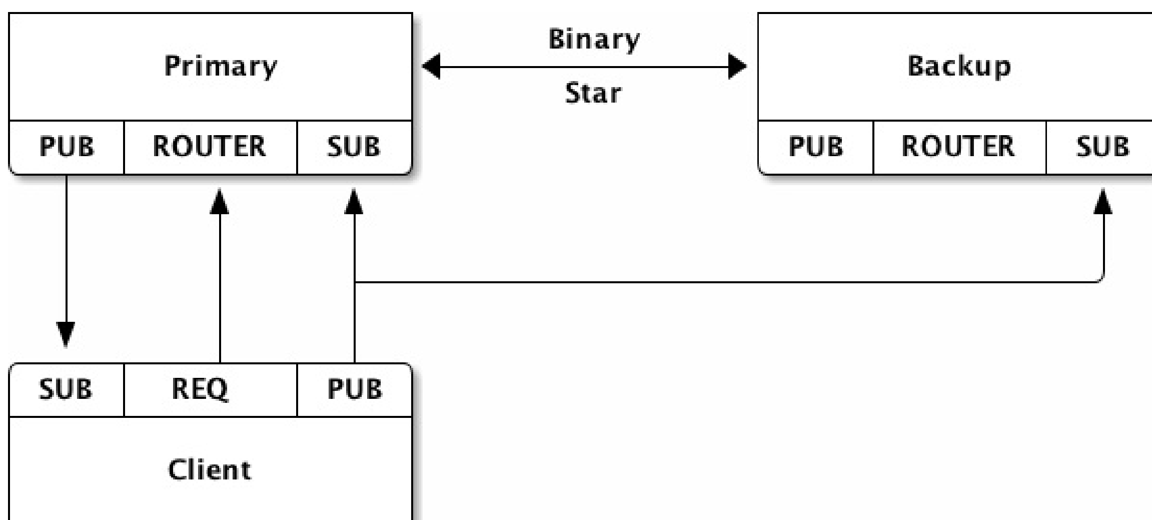


Figure 7 — High availability Clone Server Pair

开始编程之前，我们需要将客户端重构成一个可复用的类。在ZMQ中写异步类有时是为了练习如何写出优雅的代码，但这里我们确实是希望克隆模式可以成为一种易于使用的程序。上述架构的伸缩性来源于客户端的正确行为，因此有必要将其封装成一份API。要在客户端中进行故障恢复还是比较复杂的，试想一下自由者模式和克隆模式结合起来会是什么样的吧。

按照我的习惯，我会先写出一份API的列表，然后加以实现。让我们假想一个名为clone的API，在其基础之上编写克隆模式客户端API。将代码封装为API显然会提升代码的稳定性，就以模型5为例，客户端需要打开三个套接字，端点名称直接写在了代码里。我们可以创建这样一组API：

```
//  为每个套接字指定端点
clone_subscribe (clone, "tcp://localhost:5556");
clone_snapshot  (clone, "tcp://localhost:5557");
clone_updates   (clone, "tcp://localhost:5558");

//  由于有两个服务端，因此再执行一次
clone_subscribe (clone, "tcp://localhost:5566");
clone_snapshot  (clone, "tcp://localhost:5567");
clone_updates   (clone, "tcp://localhost:5568");
```

但这种写法还是比较啰嗦的，因为没有必要将API内部的一些设计暴露给编程人员。现在我们会使用三个套接字，而将来可能就会使用两个，或者四个。我们不可能让所有的应用程序都相应地修改吧？让我们把这些信息包装到API中：

```
//  指定主备服务器端点
clone_connect (clone, "tcp://localhost:5551");
clone_connect (clone, "tcp://localhost:5561");
```

这样一来代码就变得非常简洁，不过也会对现有代码的内部就够造成影响。我们需要从一个端点中推算出三个端点。一种方法是假设客户端和服务端使用三个连续的端点通信，并将这个规则写入协议；另一个方法是向服务器索取缺少的端点信息。我们使用第一种较为简单的方法：

- 服务器状态ROUTER在端点P；
- 服务器更新事件PUB在端点P + 1；
- 服务器更新事件SUB在端点P + 2。

clone类和第四章的flcliapi类很类似，由两部分组成：

- 一个在后台运行的异步克隆模式代理。该代理处理所有的I/O操作，实时地和服务器进行通信；
- 一个在前台应用程序中同步运行的clone类。当你创建了一个clone对象后，它会自动创建后台的clone线程；当你销毁clone对象，该后台线程也会被销毁。

前台的clone类会使用inproc管道和后台的代理进行通信。C语言中，czmq线程会自动为我们创建这个管道。这也是ZMQ多线程编程的常规方式。

如果没有ZMQ，这种异步的设计将很难处理高压工作，而ZMQ会让其变得简单。编写出来额代码会相对比较复杂。我们可以用反应堆的模式来编写，但这会进一步增加复杂度，且影响应用程序的使用。因此，我们的设计的API将更像是一个能够和服务器进行通信的键值表：

```
clone_t *clone_new (void);
void clone_destroy (clone_t **self_p);
void clone_connect (clone_t *self, char *address, char *service);
void clone_set (clone_t *self, char *key, char *value);
char *clone_get (clone_t *self, char *key);
```

下面就是克隆模式客户端模型6的代码，因为调用了API，所以非常简短：
clonecli6: Clone client, Model Six in C

```
//
//  克隆模式 - 客户端 - 模型6
//

//  直接编译，不建类库
#include "clone.c"

#define SUBTREE "/client/"

int main (void)
{
    //  创建分布式哈希表
    clone_t *clone = clone_new ();

    //  配置
    clone_subtree (clone, SUBTREE);
    clone_connect (clone, "tcp://localhost", "5556");
    clone_connect (clone, "tcp://localhost", "5566");

    //  插入随机键值
    while (!zctx_interrupted) {
        //  生成随机值
        char key [255];
        char value [10];
        sprintf (key, "%s%d", SUBTREE, randof (10000));
        sprintf (value, "%d", randof (1000000));
        clone_set (clone, key, value, randof (30));
        sleep (1);
    }
    clone_destroy (&clone);
    return 0;
}
```

以下是clone类的实现： **clone: Clone class in C**

```
/*  ==================================================================
    clone - client-side Clone Pattern class

    ------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it
    the terms of the GNU Lesser General Public License as published
    the Free Software Foundation; either version 3 of the License,
    your option) any later version.

    This software is distributed in the hope that it will be useful
    WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GN
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Publi
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    ===============================================================
*/

#include "clone.h"

//  请求超时时间
#define GLOBAL_TIMEOUT  4000    //  msecs
//  判定服务器死亡的时间
#define SERVER_TTL      5000    //  msecs
//  服务器数量
#define SERVER_MAX      2


//  ===============================================================
//  同步部分，在应用程序线程中工作

//  ---------------------------------------------------------------
//  类结构

struct _clone_t {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  和后台代理间的通信套接字
};

//  该线程用于处理真正的clone类
static void clone_agent (void *args, zctx_t *ctx, void *pipe);


//  ---------------------------------------------------------------
//  构造函数

clone_t *
clone_new (void)
{
    clone_t
        *self;

    self = (clone_t *) zmalloc (sizeof (clone_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, clone_agent, NULL);
    return self;
}

//  ---------------------------------------------------------------
//  析构函数

void
```

```
clone_destroy (clone_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        clone_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
//  在链接之前指定快照和更新事件的子树
//  发送给后台代理的消息内容为[SUBTREE][subtree]

void clone_subtree (clone_t *self, char *subtree)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SUBTREE");
    zmsg_addstr (msg, subtree);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
//  连接至新的服务器端点
//  消息内容：[CONNECT][endpoint][service]

void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
//  设置新值
//  消息内容：[SET][key][value][ttl]

void
clone_set (clone_t *self, char *key, char *value, int ttl)
{
    char ttlstr [10];
    sprintf (ttlstr, "%d", ttl);

    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SET");
    zmsg_addstr (msg, key);
```

```
    zmsg_addstr (msg, value);
    zmsg_addstr (msg, ttlstr);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------
//  取值
//  消息内容：[GET][key]
//  如果没有clone可用，会返回NULL

char *
clone_get (clone_t *self, char *key)
{
    assert (self);
    assert (key);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "GET");
    zmsg_addstr (msg, key);
    zmsg_send (&msg, self->pipe);

    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *value = zmsg_popstr (reply);
        zmsg_destroy (&reply);
        return value;
    }
    return NULL;
}


//  ===============================================================
//  异步部分，在后台运行

//  ---------------------------------------------------------------
//  单个服务端信息

typedef struct {
    char *address;              //  服务端地址
    int port;                   //  端口
    void *snapshot;             //  快照套接字
    void *subscriber;           //  接收更新事件的套接字
    uint64_t expiry;            //  服务器过期时间
    uint requests;              //  收到的快照请求数
} server_t;

static server_t *
server_new (zctx_t *ctx, char *address, int port, char *subtree)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));

    zclock_log ("I: adding server %s:%d...", address, port);
    self->address = strdup (address);
    self->port = port;
```

```
    self->snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (self->snapshot, "%s:%d", address, port);
    self->subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "%s:%d", address, port + 1);
    zsockopt_set_subscribe (self->subscriber, subtree);
    return self;
}

static void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->address);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
//  后台代理类

//  状态
#define STATE_INITIAL       0   //  连接之前
#define STATE_SYNCING       1   //  正在同步
#define STATE_ACTIVE        2   //  正在更新

typedef struct {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  与主线程通信的套接字
    zhash_t *kvmap;             //  键值表
    char *subtree;              //  子树
    server_t *server [SERVER_MAX];
    uint nbr_servers;           //  范围：0 - SERVER_MAX
    uint state;                 //  当前状态
    uint cur_server;            //  当前master，0/1
    int64_t sequence;           //  键值对编号
    void *publisher;            //  发布更新事件的套接字
} agent_t;

static agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->kvmap = zhash_new ();
    self->subtree = strdup ("");
    self->state = STATE_INITIAL;
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    return self;
```

```
}

static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        int server_nbr;
        for (server_nbr = 0; server_nbr < self->nbr_servers; server
            server_destroy (&self->server [server_nbr]);
        zhash_destroy (&self->kvmap);
        free (self->subtree);
        free (self);
        *self_p = NULL;
    }
}

//  若线程被中断则返回-1
static int
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);
    if (command == NULL)
        return -1;

    if (streq (command, "SUBTREE")) {
        free (self->subtree);
        self->subtree = zmsg_popstr (msg);
    }
    else
    if (streq (command, "CONNECT")) {
        char *address = zmsg_popstr (msg);
        char *service = zmsg_popstr (msg);
        if (self->nbr_servers < SERVER_MAX) {
            self->server [self->nbr_servers++] = server_new (
                self->ctx, address, atoi (service), self->subtree);
            //  广播更新事件
            zsocket_connect (self->publisher, "%s:%d",
                address, atoi (service) + 2);
        }
        else
            zclock_log ("E: too many servers (max. %d)", SERVER_MAX
        free (address);
        free (service);
    }
    else
    if (streq (command, "SET")) {
        char *key = zmsg_popstr (msg);
        char *value = zmsg_popstr (msg);
        char *ttl = zmsg_popstr (msg);
        zhash_update (self->kvmap, key, (byte *) value);
```

```
        zhash_freefn (self->kvmap, key, free);

        //  向服务端发送键值对
        kvmsg_t *kvmsg = kvmsg_new (0);
        kvmsg_set_key  (kvmsg, key);
        kvmsg_set_uuid (kvmsg);
        kvmsg_fmt_body (kvmsg, "%s", value);
        kvmsg_set_prop (kvmsg, "ttl", ttl);
        kvmsg_send     (kvmsg, self->publisher);
        kvmsg_destroy (&kvmsg);
puts (key);
        free (ttl);
        free (key);             //  键值对实际由哈希表对象控制
    }
    else
    if (streq (command, "GET")) {
        char *key = zmsg_popstr (msg);
        char *value = zhash_lookup (self->kvmap, key);
        if (value)
            zstr_send (self->pipe, value);
        else
            zstr_send (self->pipe, "");
        free (key);
        free (value);
    }
    free (command);
    zmsg_destroy (&msg);
    return 0;
}


//  ---------------------------------------------------------------------
//  异步的后台代理会维护一个服务端池，并处理来自应用程序的请求或应答。

static void
clone_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    while (TRUE) {
        zmq_pollitem_t poll_set [] = {
            { pipe, 0, ZMQ_POLLIN, 0 },
            { 0,    0, ZMQ_POLLIN, 0 }
        };
        int poll_timer = -1;
        int poll_size = 2;
        server_t *server = self->server [self->cur_server];
        switch (self->state) {
            case STATE_INITIAL:
                //  该状态下，如果有可用服务，会发送快照请求
                if (self->nbr_servers > 0) {
                    zclock_log ("I: 正在等待服务器 %s:%d...",
                        server->address, server->port);
```

```
                if (server->requests < 2) {
                    zstr_sendm (server->snapshot, "ICANHAZ?");
                    zstr_send  (server->snapshot, self->subtree
                    server->requests++;
                }
                server->expiry = zclock_time () + SERVER_TTL;
                self->state = STATE_SYNCING;
                poll_set [1].socket = server->snapshot;
            }
            else
                poll_size = 1;
            break;
        case STATE_SYNCING:
            //  该状态下我们从服务器端接收快照内容，若失败则尝试其他服务
            poll_set [1].socket = server->snapshot;
            break;
        case STATE_ACTIVE:
            //  该状态下我们从服务器获取更新事件，失败则尝试其他服务器
            poll_set [1].socket = server->subscriber;
            break;
    }
    if (server) {
        poll_timer = (server->expiry - zclock_time ())
                   * ZMQ_POLL_MSEC;
        if (poll_timer < 0)
            poll_timer = 0;
    }
    //  ---------------------------------------------------------
    //  poll循环
    int rc = zmq_poll (poll_set, poll_size, poll_timer);
    if (rc == -1)
        break;              //  上下文已被关闭

    if (poll_set [0].revents & ZMQ_POLLIN) {
        if (agent_control_message (self))
            break;          //  中断
    }
    else
    if (poll_set [1].revents & ZMQ_POLLIN) {
        kvmsg_t *kvmsg = kvmsg_recv (poll_set [1].socket);
        if (!kvmsg)
            break;          //  中断

        //  任何服务端的消息将重置它的过期时间
        server->expiry = zclock_time () + SERVER_TTL;
        if (self->state == STATE_SYNCING) {
            //  保存快照内容
            server->requests = 0;
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                self->state = STATE_ACTIVE;
                zclock_log ("I: received from %s:%d snapshot=%d
                    server->address, server->port,
```

```
                        (int) self->sequence);
                    kvmsg_destroy (&kvmsg);
                }
                else
                    kvmsg_store (&kvmsg, self->kvmap);
            }
            else
            if (self->state == STATE_ACTIVE) {
                //  丢弃过期的更新事件
                if (kvmsg_sequence (kvmsg) > self->sequence) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, self->kvmap);
                    zclock_log ("I: received from %s:%d update=%d",
                        server->address, server->port,
                        (int) self->sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
        }
        else {
            //  服务端已死，尝试其他服务器
            zclock_log ("I: 服务器 %s:%d 无响应",
                    server->address, server->port);
            self->cur_server = (self->cur_server + 1) % self->nbr_s
            self->state = STATE_INITIAL;
        }
    }
    agent_destroy (&self);
}
```

最后是克隆服务器的模型6代码：

clonesrv6: Clone server, Model Six in C

```
//
// 克隆模式 - 服务端 - 模型6
//

//  直接编译，不建类库
#include "bstar.c"
#include "kvmsg.c"

//  bstar反应堆API
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);
static int s_send_hugz  (zloop_t *loop, void *socket, void *args);
static int s_new_master (zloop_t *loop, void *unused, void *args);
static int s_new_slave  (zloop_t *loop, void *unused, void *args);
static int s_subscriber (zloop_t *loop, void *socket, void *args);
```

```
//  服务端属性
typedef struct {
    zctx_t *ctx;                //  上下文
    zhash_t *kvmap;             //  存放键值对
    bstar_t *bstar;             //  bstar反应堆核心
    int64_t sequence;           //  更新事件编号
    int port;                   //  主端口
    int peer;                   //  同伴端口
    void *publisher;            //  发布更新事件的端口
    void *collector;            //  接收客户端更新事件的端口
    void *subscriber;           //  接受同伴更新事件的端口
    zlist_t *pending;           //  延迟的更新事件
    Bool primary;               //  是否为主机
    Bool master;                //  是否为master
    Bool slave;                 //  是否为slave
} clonesrv_t;


int main (int argc, char *argv [])
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    if (argc == 2 && streq (argv [1], "-p")) {
        zclock_log ("I: 作为主机master运行，正在等待备机slave连接。");
        self->bstar = bstar_new (BSTAR_PRIMARY, "tcp://*:5003",
                                 "tcp://localhost:5004");
        bstar_voter (self->bstar, "tcp://*:5556", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5556;
        self->peer = 5566;
        self->primary = TRUE;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        zclock_log ("I: 作为备机slave运行，正在等待主机master连接。");
        self->bstar = bstar_new (BSTAR_BACKUP, "tcp://*:5004",
                                 "tcp://localhost:5003");
        bstar_voter (self->bstar, "tcp://*:5566", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5566;
        self->peer = 5556;
        self->primary = FALSE;
    }
    else {
        printf ("Usage: clonesrv4 { -p | -b }\n");
        free (self);
        exit (0);
    }
    //  主机将成为master
    if (self->primary)
        self->kvmap = zhash_new ();

    self->ctx = zctx_new ();
```

```
    self->pending = zlist_new ();
    bstar_set_verbose (self->bstar, TRUE);

    //  设置克隆服务端套接字
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  作为克隆客户端连接同伴
    self->subscriber = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "tcp://localhost:%d", self->

    //  注册状态事件处理器
    bstar_new_master (self->bstar, s_new_master, self);
    bstar_new_slave (self->bstar, s_new_slave, self);

    //  注册bstar反应堆其他事件处理器
    zloop_reader (bstar_zloop (self->bstar), self->collector, s_col
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_flush_ttl,
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_send_hugz,

    //  开启bstar反应堆
    bstar_start (self->bstar);

    //  中断，终止。
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_destroy (&kvmsg);
    }
    zlist_destroy (&self->pending);
    bstar_destroy (&self->bstar);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);

    return 0;
}


//  ---------------------------------------------------------------
//  发送快照内容

static int s_send_single (char *key, void *data, void *args);

//  请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求放标识
    char *subtree;          //  子树
} kvroute_t;

static int
```

```
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  请求在消息的第二帧中
        char *request = zstr_recv (snapshot);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: 错误的请求，正在退出……\n");

        if (subtree) {
            //  发送状态快照
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  发送终止消息，以及消息编号
            zclock_log ("I: 正在发送快照，版本号：%d", (int) self->seq
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}


//  每次发送一个快照键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0)
        //  先发送接收方的地址
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

```
//  ---------------------------------------------------------------------
//  从客户端收集更新事件
//  如果我们是master，则将该事件写入kvmap对象；
//  如果我们是slave，则将其写入延迟队列

static int s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg);

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (collector);
    kvmsg_dump (kvmsg);
    if (kvmsg) {
        if (self->master) {
            kvmsg_set_sequence (kvmsg, ++self->sequence);
            kvmsg_send (kvmsg, self->publisher);
            int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
            if (ttl)
                kvmsg_set_prop (kvmsg, "ttl",
                    "%" PRId64, zclock_time () + ttl * 1000);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: 正在发布更新事件：%d", (int) self->sequenc
        }
        else {
            //  如果我们已经从master中获得了该事件，则丢弃该消息
            if (s_was_pending (self, kvmsg))
                kvmsg_destroy (&kvmsg);
            else
                zlist_append (self->pending, kvmsg);
        }
    }
    return 0;
}

//  如果消息已在延迟队列中，则删除它并返回TRUE

static int
s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg)
{
    kvmsg_t *held = (kvmsg_t *) zlist_first (self->pending);
    while (held) {
        if (memcmp (kvmsg_uuid (kvmsg),
                    kvmsg_uuid (held), sizeof (uuid_t)) == 0) {
            zlist_remove (self->pending, held);
            return TRUE;
        }
        held = (kvmsg_t *) zlist_next (self->pending);
    }
    return FALSE;
}
```

```
//  ---------------------------------------------------------------------
//  删除带有过期时间的瞬间值

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  如果键值对过期，则进行删除操作，并广播该事件
static int
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布删除事件：%d", (int) self->sequence);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  发送心跳

static int
s_send_hugz (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_new (self->sequence);
    kvmsg_set_key  (kvmsg, "HUGZ");
    kvmsg_set_body (kvmsg, (byte *) "", 0);
    kvmsg_send     (kvmsg, self->publisher);
    kvmsg_destroy (&kvmsg);

    return 0;
}
```

```
//  ---------------------------------------------------------------------
//  状态改变事件处理函数
//  我们将转变为master
//
//  备机先将延迟列表中的事件更新到自己的快照中，
//  并开始接收客户端发来的快照请求。

static int
s_new_master (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    self->master = TRUE;
    self->slave = FALSE;
    zloop_cancel (bstar_zloop (self->bstar), self->subscriber);

    //  应用延迟列表中的事件
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布延迟列表中的更新事件：%d", (int) self->s
    }
    return 0;
}

//  ---------------------------------------------------------------------
//  正在切换为slave

static int
s_new_slave (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zhash_destroy (&self->kvmap);
    self->master = FALSE;
    self->slave = TRUE;
    zloop_reader (bstar_zloop (self->bstar), self->subscriber,
                  s_subscriber, self);

    return 0;
}

//  ---------------------------------------------------------------------
//  从同伴主机（master）接收更新事件；
//  接收该类更新事件时，我们一定是slave。

static int
s_subscriber (zloop_t *loop, void *subscriber, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    //  获取快照，如果需要的话。
```

```
    if (self->kvmap == NULL) {
        self->kvmap = zhash_new ();
        void *snapshot = zsocket_new (self->ctx, ZMQ_DEALER);
        zsocket_connect (snapshot, "tcp://localhost:%d", self->peer
        zclock_log ("I: 正在请求快照：tcp://localhost:%d",
                    self->peer);
        zstr_send (snapshot, "ICANHAZ?");
        while (TRUE) {
            kvmsg_t *kvmsg = kvmsg_recv (snapshot);
            if (!kvmsg)
                break;          //  中断
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_destroy (&kvmsg);
                break;          //  完成
            }
            kvmsg_store (&kvmsg, self->kvmap);
        }
        zclock_log ("I: 收到快照，版本号：%d", (int) self->sequence);
        zsocket_destroy (self->ctx, snapshot);
    }
    //  查找并删除
    kvmsg_t *kvmsg = kvmsg_recv (subscriber);
    if (!kvmsg)
        return 0;

    if (strneq (kvmsg_key (kvmsg), "HUGZ")) {
        if (!s_was_pending (self, kvmsg)) {
            //  如果master的更新事件比客户端的事件早到，则将master的事件存
            //  当收到客户端更新事件时会将其从列表中清除。
            zlist_append (self->pending, kvmsg_dup (kvmsg));
        }
        //  如果更新事件比kvmap版本高，则应用它
        if (kvmsg_sequence (kvmsg) > self->sequence) {
            self->sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: 收到更新事件：%d", (int) self->sequence);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    else
        kvmsg_destroy (&kvmsg);

    return 0;
}
```

这段程序只有几百行，但还是花了一些时间来进行调通的。这个模型中包含了故障恢复，瞬间值，子树等等。虽然我们前期设计得很完备，但要在多个套接字之间进行调试还是很困难的。以下是我的工作方式：

- 由于使用了反应堆（bstar，建立在zloop之上），我们节省了大量的代码，让程序变得简洁明了。整个服务以一个线程运行，因此不会出现跨线程的问题。只需将结构指针（self）传递给所有的处理器即可。此外，使用发应堆后可以让代码更为模块化，易于重用。

- 我们逐个模块进行调试，只有某个模块能够正常运行时才会进入下一步。由于使用了四五个套接字，因此调试的工作量是很大的。我直接将调试信息输出到了屏幕上，因为实在没有必要专门开一个调试器来工作。

- 因为一直在使用valgrind工具进行测试，因此我能确定程序没有内存泄漏的问题。在C语言中，内存泄漏是我们非常关心的问题，因为没有什么垃圾回收机制可以帮你完成。正确地使用像kvmsg、czmq之类的抽象层可以很好地避免内存泄漏。

这段程序肯定还会存在一些BUG，部分读者可能会帮助我调试和修复，我在此表示感谢。

测试模型6时，先开启主机和备机，再打开一组客户端，顺序随意。随机地中止某个服务进程，如果程序设计得是正确的，那客户端获得的数据应该都是一致的。

克隆模式协议

花费了那么多精力来开发一套可靠的发布-订阅模式机制，我们当然希望将来能够方便地在其基础之上进行扩展。较好的方法是将其编写为一个协议，这样就能让各种语言来实现它了。

我们将其称为“集群化哈希表协议”，这是一个能够跨集群地进行键值哈希表管理，提供了多客户端的通信机制；客户端可以只操作一个子树的数据，包括更新和定义瞬间值。

- http://rfc.zeromq.org/spec:12